USENIX

Proceedings of the 17th USENIX Security Symposium

San Jose, CA, USA  July 28–August 1, 2008

conference

proceedings

# 17th USENIX Security Symposium

*San Jose, CA, USA*

*July 28–August 1, 2008*

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $45 for members and $55 for nonmembers.
Outside the U.S.A. and Canada, please add $20 per copy for postage (via air printed matter).

## Thanks to Our Sponsors

### SILVER SPONSOR

Google™

### SPONSORS

BAE SYSTEMS

BARRACUDA NETWORKS

IBM Research

Dartmouth College
INSTITUTE FOR SECURITY TECHNOLOGY STUDIES

NETWITNESS

## Thanks to Our Media Sponsors

ACM *Queue*
Addison-Wesley Professional/ Prentice Hall Professional/ Cisco Press
*BetaNews*
*Dr. Dobb's Journal*
*Free Software Magazine*
*Homeland Defense Journal*
*IEEE Security & Privacy*
*InfoSec News*
ITtoolbox
*Linux+DVD Magazine*
*Linux Gazette*
*Linux Journal*
*Linux Pro Magazine*
LXer.com
No Starch Press
StorageNetworking.org
*The Register*
UserFriendly.org

ISBN-13: 978-1-931971-60-7

USENIX Association

# Proceedings of the 17th USENIX Security Symposium

July 28–August 1, 2008
San Jose, CA, USA

## Conference Organizers

### Program Chair
Paul Van Oorschot, *Carleton University*

### Program Committee
Paul Barford, *University of Wisconsin*
Dan Boneh, *Stanford University*
Bill Cheswick, *AT&T Labs—Research*
David Dagon, *Georgia Institute of Technology*
George Danezis, *Microsoft Research*
Rachna Dhamija, *Harvard University*
Virgil Gligor, *Carnegie Mellon University*
Ian Goldberg, *University of Waterloo*
Trent Jaeger, *Pennsylvania State University*
Somesh Jha, *University of Wisconsin*
Angelos Keromytis, *Columbia University*
Tadayoshi Kohno, *University of Washington*
Wenke Lee, *Georgia Institute of Technology*
David Lie, *University of Toronto*
Z. Morley Mao, *University of Michigan*
Fabian Monrose, *Johns Hopkins University*
Michael Reiter, *University of North Carolina*
R. Sekar, *Stony Brook University*
Micah Sherr, *University of Pennsylvania*
Anil Somayaji, *Carleton University*
Angelos Stavrou, *George Mason University*
Patrick Traynor, *Pennsylvania State University*
Leendert van Doorn, *AMD*
Wietse Venema, *IBM Research*
Xiaolan (Catherine) Zhang, *IBM Research*

### Invited Talks Committee
Bill Aiello, *University of British Columbia*
Angelos Keromytis, *Columbia University*
Avi Rubin, *Johns Hopkins University*

### Poster Session Chair
Carrie Gates, *CA Labs*

### Work-in-Progress Reports Chair
Hao Chen, *University of California, Davis*

### The USENIX Association Staff

## External Reviewers

Michael Abd-El-Malek
Adam Aviv
Farshad Bahari
Lucas Ballard
Adam Barth
Steve Bellovin
Matthew Burnside
Kevin Butler
Martim Carbone
Lorenzo Cavallaro
Pavol Cerny
Pau-Chen Cheng
Mihai Christodorescu
Sandy Clark
Mauro Conti
Scott Coull
Gabriela Cretu
Eric Cronin
Sven Dietrich
Valden Djeric
William Enck
David Evans
Scott Garriss
Sam Guarnieri
Eric Hall
Mike Hamburg
Tom Hart
Michael Hohmuth
Collin Jackson
Shvetank Jain
Guruprasad Jakka
Stanislaw Jarecki
Xuxian Jiang
Emilia Kasper
David H. King
Louis Kruger
Andrea Lanzi
Soo Bum Lee
Weijen Lee
Lionel Litty
Michael Locasto
Mohammad Mannan
Lorenzo Martignoni
Damon McCoy
Stephen McLaughlin
Divya Muthukumaran
Bryan Payne
Moheeb Rajab
Sandra Rueda
Luke St. Clair
Prateek Saxena
Joshua Schiffman
Gaurav Shah
Monirul Sharif
Ji Sun Shin
Kapil Singh
Yingbo Song
Vugranam Sreedhar
Yogesh Sreenivasan
Liz Stinson
Weiqing Sun
Gelareh Taban
Julie Thorpe
Alok Tongaonkar
V.N. Venkatakrishnan
Hayawardh Vijayakumar
Jiang Wang
Zhaohui Wang
Charles Wright
Wei Xu
Junjie Zhang
Lei Zhang

# 17th USENIX Security Symposium
# July 28–August 1, 2008
# San Jose, CA, USA

## Thursday, July 31

## Friday, August 1

# Index of Authors

# Message from the Program Chair

It is with great pleasure that I welcome you to the 17th USENIX Security Symposium, in San Jose, California. A total of 174 research papers were submitted to the technical program. Four were withdrawn or otherwise summarily rejected, leaving 170 to consider. From these, 27 were selected—a record number for USENIX Security—for an acceptance rate of 15.9%. Because of the very high quality of submissions, many excellent papers were not among those accepted. Some of these will no doubt be viewed as major contributions when they appear at other venues, and we hope that the detailed technical review comments we provided are helpful. I thank all authors who submitted papers. High-quality submissions are the starting point for a superb symposium.

Our Program Committee (PC) meeting was graciously hosted April 3–4 by Angelos Keromytis at Columbia University, with help from Sophie Majewski. The dinner provided afterwards by USENIX, at Pisticci Restaurant, was most welcome. As expected, the meeting was attended by essentially the entire committee—25 of 26 members. Each paper received at least three written reviews, and many were read and discussed by a considerably larger group. PC members were restricted to being co-authors on at most two submissions. The physical-presence PC meeting and the relatively small Program Committee, as is traditional for USENIX Security (in sharp contrast to several other major security research conferences), contributed to a collegial process and open discussions, pooling the expertise of the entire committee. I am immensely grateful to the committee for their cooperative spirit and extraordinary efforts. Every member delivered every review requested, and more. It was a true privilege to work with such a dedicated and focused team, many of whom seem to serve continuous tours of Program Committee duty. I extend my thanks to all of the external reviewers relied upon by the PC members; their names are recorded in the frontmatter of these proceedings.

Beyond the technical program in these proceedings, the symposium is enriched by many other items. These include two days of tutorials by area experts (prior to the technical program) and, in parallel with the submitted papers, an exceptional invited talks track. For the latter, thanks are due to our invited talks committee of Bill Aiello, Angelos Keromytis, and Avi Rubin. This year's keynote address is by Debra Bowen, California Secretary of State. I thank Carrie Gates for organizing the poster session and Hao Chen for chairing the work-in-progress reports.

Rather than repeat past praises about the USENIX staff, let me give the following advice to all potential future program chairs: if given a choice between chairing a USENIX conference and another, choose the former. You will truly understand why only once you have done both. I am happy to thank Anne Dickison for driving publicity, Jane-Ellen Long for logistics related to the proceedings, Devon Shaw for support related to the PC meeting and the conference itself, Casey Henderson for updates to the USENIX Web site, and Peter Collinson for manning the submissions system and review Web site. Did I mention Ellie Young? Who is it that pulls all the strings and holds everything together? Thanks, Ellie: it is a pleasure to work with you. Thanks also to Niels Provos, last year's Program Chair, for guidance, and to Matt Blaze for talking me into acting as this year's Program Chair.

I hope you enjoy the symposium as much as I have enjoyed being part of delivering it.

**Paul Van Oorschot, Carleton University**
**Program Chair**

# All Your `iFRAME`s Point to Us

*Niels Provos* *Panayiotis Mavrommatis*
*Google Inc.*
{niels, panayiotis}@google.com

*Moheeb Abu Rajab* *Fabian Monrose*
*Johns Hopkins University*
{moheeb, fabian}@cs.jhu.edu

## Abstract

As the web continues to play an ever increasing role in information exchange, so too is it becoming the prevailing platform for infecting vulnerable hosts. In this paper, we provide a detailed study of the pervasiveness of so-called *drive-by downloads* on the Internet. Drive-by downloads are caused by `URL`s that attempt to exploit their visitors and cause malware to be installed and run automatically. Over a period of 10 months we processed billions of `URL`s, and our results shows that a non-trivial amount, of over 3 million malicious `URL`s, initiate drive-by downloads. An even more troubling finding is that approximately 1.3% of the incoming search queries to Google's search engine returned at least one `URL` labeled as malicious in the results page. We also explore several aspects of the drive-by downloads problem. Specifically, we study the relationship between the user browsing habits and exposure to malware, the techniques used to lure the user into the malware distribution networks, and the different properties of these networks.

## 1 Introduction

It should come as no surprise that our increasing reliance on the Internet for many facets of our daily lives (*e.g.,* commerce, communication, entertainment, etc.) makes the Internet an attractive target for a host of illicit activities. Indeed, over the past several years, Internet services have witnessed major disruptions from attacks, and the network itself is continually plagued with malfeasance [14]. While the monetary gains from the myriad of illicit behaviors being perpetrated today (*e.g.,* phishing, spam) is just barely being understood [11], it is clear that there is a general shift in tactics—wide-scale attacks aimed at overwhelming computing resources are becoming less prevalent, and instead, traditional scanning attacks are being replaced by other mechanisms. Chief among these is the exploitation of the web, and the services built upon it, to distribute malware.

This change in the playing field is particularly alarming, because unlike traditional scanning attacks that use push-based infection to increase their population, web-based malware infection follows a pull-based model. For the most part, the techniques in use today for delivering web-malware can be divided into two main categories. In the first case, attackers use various social engineering techniques to entice the visitors of a website to download and run malware. The second, more devious case, involves the underhanded tactic of targeting various browser vulnerabilities to *automatically* download and run—i.e., unknowingly to the visitor—the binary upon visiting a website. When popular websites are exploited, the potential victim base from these so-called *drive-by downloads* can be far greater than other forms of exploitation because traditional defenses (e.g., firewalls, dynamic addressing, proxies) pose no barrier to infection. While social engineering may, in general, be an important malware spreading vector, in this work we restrict our focus and analysis to malware delivered via drive-by downloads.

Recently, Provos *et al*. [20] provided insights on this new phenomenon, and presented a cursory overview of web-based malware. Specifically, they described a number of server- and client-side exploitation techniques that are used to spread malware, and elucidated the mechanisms by which a successful exploitation chain can start and continue to the automatic installation of malware. In this paper, we present a detailed analysis of the malware serving infrastructure on the web using a large corpus of malicious URLs collected over a period of ten months. Using this data, we estimate the global prevalence of drive-by downloads, and identify several trends for different aspects of the web malware problem. Our results reveal an alarming contribution of Chinese-based web sites to the web malware problem: overall, 67% of the malware distribution servers and 64% of the web sites that link to them are located in China. These results raise serious question about the security practices employed by web site administrators.

Additionally, we study several properties of the malware serving infrastructure, and show that (for the most part) the malware serving networks are composed of tree-like structures with strong fan-in edges leading to the main malware distribution sites. These distribution sites normally deliver the malware to the victim after a number of indirection steps traversing a path on the distribution network tree. More interestingly, we show that several malware distribution networks have linkages that can be attributed to various relationships.

In general, the edges of these malware distribution networks represent the hop-points used to lure users to the malware distribution site. By investigating these edges, we reveal a number of causal relationships that eventually lead to browser exploitation. More troubling, we show that drive-by downloads are being induced by mechanisms beyond the conventional techniques of controlling the content of compromised websites. In particular, our results reveal that Ad serving networks are increasingly being used as hops in the malware serving chain. We attribute this increase to syndication, a common practice which allows advertisers to rent out part of their advertising space to other parties. These findings are problematic as they show that even protected webservers can be used as vehicles for transferring malware. Additionally, we also show that contrary to common wisdom, the practice of following "safe browsing" habits (*i.e.,* avoiding gray content) by itself is not an effective safeguard against exploitation.

The remainder of this paper is organized as follows. In Section 2, we provide background information on how vulnerable computer systems can be compromised solely by visiting a malicious web page. Section 3 gives an overview of our data collection infrastructure and in Section 4 we discuss the prevalence of malicious web sites on the Internet. In Section 5, we explore the mechanisms used to inject malicious content into web pages. We analyze several aspects of the web malware distribution networks in Section 6. In Section 7 we provide an overview of the impact of the installed malware on the infected system. Section 8 discusses implications of our results and Section 9 presents related work. Finally, we conclude in Section 10.

## 2 Background

Unfortunately, there are a number of existing exploitation strategies for installing malware on a user's computer. One common technique for doing so is by remotely exploiting vulnerable network services. However, lately, this attack strategy has become less successful (and presumably, less profitable). Arguably, the proliferation of technologies such as Network Address Translators (NATs) and firewalls make it difficult to remotely connect and exploit services running on users' computers. This, in turn, has lead attackers to seek other avenues of exploitation. An equally potent alternative is to simply lure web users to connect to (compromised) malicious servers that subsequently deliver exploits targeting vulnerabilities of web browsers or their plugins.

Adversaries use a number of techniques to inject content under their control into benign websites. In many cases, adversaries exploit web servers via vulnerable scripting applications. Typically, these vulnerabilities (*e.g.,* in phpBB2 or InvisionBoard) allow an adversary to gain direct access to the underlying operating system. That access can often be escalated to super-user privileges which in turn can be used to compromise any web server running on the compromised host. In general, upon successful exploitation of a web server the adversary injects new content to the compromised website. In most cases, the injected content is a link that redirects the visitors of these websites to a `URL` that hosts a script crafted to exploit the browser. To avoid visual detection by website owners, adversaries normally use invisible HTML components (*e.g.,* zero pixel `IFRAME`s) to hide the injected content.

Another common content injection technique is to use websites that allow users to contribute their own content, for example, via postings to forums or blogs. Depending on the site's configuration, user contributed content may be restricted to text but often can also contain HTML such as links to images or other external content. This is particularly dangerous, as without proper filtering in place, the adversary can simply inject the exploit `URL` without the need to compromise the web server.

Figure 1 illustrates the main phases in a typical interaction that takes place when a user visits a website with injected malicious content. Upon visiting this website, the browser downloads the initial exploit script (*e.g.,* via an `IFRAME`). The exploit script (in most cases, `javascript`) targets a vulnerability in the browser or one of its plugins. Interested readers are referred to Provos *et al.* [20] for a number of vulnerabilities that are commonly used to gain control of the infected system. Successful exploitation of one of these vulnera-

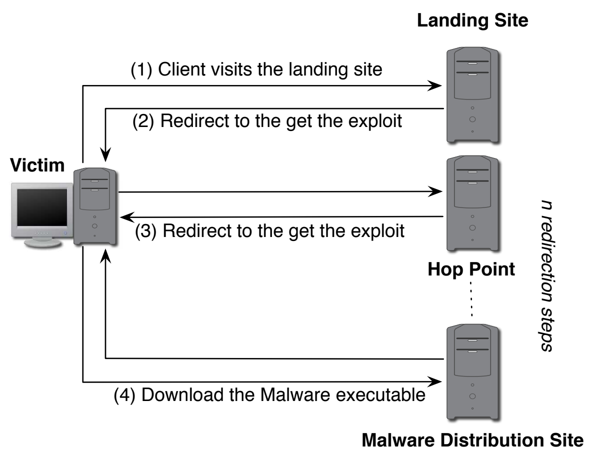

Landing Site
(1) Client visits the landing site
(2) Redirect to the get the exploit
Victim
(3) Redirect to the get the exploit
Hop Point
n redirection steps
(4) Download the Malware executable
Malware Distribution Site

Figure 1: A typical Interaction with of drive-by download victim with a landing URL .

bilities results in the automatic execution of the exploit code, thereby triggering a drive-by download. Drive-by downloads start when the exploit instructs the browser to connect to a malware distribution site to retrieve malware executable(s). The downloaded executable is then automatically installed and started on the infected system[1].

Finally, attackers use a number of techniques to evade detection and complicate forensic analysis. For example, the use of randomly seeded obfuscated `javascript` in their exploit code is not uncommon. Moreover, to complicate network based detection attackers use a number or redirection steps before the browser eventually contacts the malware distribution site.

## 3 Infrastructure and Methodology

Our primary objective is to identify malicious web sites (*i.e.*, URLs that trigger drive-by downloads) and help improve the safety of the Internet. Before proceeding further with the details of our data collection methodology, we first define some terms we use throughout this paper. We use the terms *landing pages* and *malicious* URLs interchangeably to denote the URLs that initiate drive-by downloads when users visit them. In our subsequent analysis, we group these URLs according to their top level domain names and we refer to the resulting set as the *landing sites*. In many cases, the malicious payload is not hosted on the landing site, but instead loaded via an IFRAME or a SCRIPT from a remote site. We call the remote site that hosts malicious payloads a *distribution site*. In what follows, we detail the different components of our data collection infrastructure.

**Pre-processing Phase.** As Figure 2 illustrates, the data processing starts from a large web repository maintained by Google. Our goal is to inspect URLs from this repository and identify the ones that trigger drive-by downloads. However, exhaustive inspection of each URL in the repository is prohibitively expensive due to the large number of URLs in the repository (on the order of billions). Therefore, we first use light-weight techniques to extract URLs that are likely malicious then subject them to a more detailed analysis and verification phase.

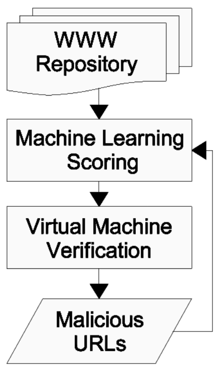


Figure 2: URL selection and verification workflow.

We employ the *mapreduce* [9] framework to process billions of web pages in parallel. For each web page, we extract several features, some of which take advantage of the fact that many landing URLs are hijacked to include malicious payload(s) or to point to malicious payload(s) from a distribution site. For example, we use "out of place" IFRAMEs, obfuscated JavaScript, or IFRAMEs to known distribution sites as features. Using a specialized machine-learning framework [7], we translate these features into a likelihood score. We employ five-fold cross-validation to measure the quality of the machine-learning framework. The cross-validation operates by splitting the data set into 5 randomly chosen partitions and then training on four partitions while using the remaining partition for validation. This process is repeated five times. For each trained model, we create an ROC curve and use the average ROC curve to estimate the overall accuracy. Using this ROC curve, we estimate the false positive and detection rate for different thresholds. Our infrastructure pre-processes roughly *one billion* pages daily. In order to fully utilize the capacity of the subsequent detailed verification phase, we choose a threshold score that results in an outcome false positive rate of about $10^{-3}$ with a corresponding detection rate of approximately $0.9$. This amounts to about one million URLs that we subject to the computationally more expensive verification phase.

In addition to analyzing web pages in the crawled web repository, we also regularly select several hundred thousands URLs for in-depth verification. These URLs are randomly sampled from popular URLs as well as from the global index. We also process URLs reported by users.

**Verification Phase.** This phase aims to verify whether a candidate URL from the pre-processing phase is malicious (*i.e.,* initiates a drive-by download). To do that, we developed a large scale *web-honeynet* that simultaneously runs a large number of Microsoft Windows images in virtual machines. Our system design draws on the experience from earlier work [25], and includes unique features that are specific to our goals. In what follows we discuss the details of the URL verification process.

Each honeypot instance runs an unpatched version of Internet Explorer. To inspect a candidate URL , the system first loads a clean Windows image then automatically starts the browser and instructs it to visit the candidate URL . We detect malicious URLs using a combination of execution based heuristics and results from anti-virus engines. Specifically, for each visited URL we run the virtual machine for approximately two minutes and monitor the system behavior for abnormal state changes including file system changes, newly created processes and changes to the system's registry. Additionally, we subject the HTTP responses to virus scans using multiple anti-virus engines. To detect malicious URLs , we develop scoring heuristics used to determines the likelihood that a URL is malicious. We determine a URL score based on a combined measure of the different state changes resulting from visiting the URL . Our heuristics score URLs based on the number of created processes, the number of observed registry changes and the number of file system changes resulting from visiting the URL .

To limit false positives, we choose a conservative decision criteria that uses an empirically derived threshold to mark a URL as malicious. This threshold is set such that it will be met if we detect changes in the system state, including the file system as well as creation of new processes. A visited URL is marked as *malicious* if it meets the threshold *and* one of the incoming HTTP responses is marked as malicious by at least one anti-virus scanner. Our extensive evaluation shows that this criteria introduces negligible false positives. Finally, a URL that meets the threshold requirement but has no incoming payload flagged by any of the anti-virus engines, is marked as *suspicious*.

On average, the detailed verification stage processes about one million URLs daily, of which roughly $25,000$ new URLs are flagged as malicious. The verification system records all the network interactions as well as the state changes. In what follows, we describe how we process the network traces associated with the detected malicious URLs to shed light on the malware distribution infrastructure.

**Constructing the Malware Distribution Networks.** To understand the properties of the web malware serving infrastructure on the Internet, we analyze the recorded network traces associated with the detected malicious URLs to construct the *malware distribution networks*. We define a distribution network as the set of malware delivery trees from all the landing sites that lead to a particular malware distribution site. A malware delivery tree consists of the landing site, as the leaf node, and all nodes (*i.e.,* web sites) that the browser visits until it contacts the malware distribution site (the root of the tree). To construct the delivery trees we extract the edges connecting these nodes by inspecting the Referer header from the recorded successive HTTP requests the browser makes after visiting the landing page. However, in many cases the Referer headers are not sufficient to extract the full chain. For example, when the browser redirection results from an external script the Referrer, in this case, points to the base page and not the external script file. Additionally, in many cases the Referer header is not set (*e.g.,* because the requests are made from within a browser plugin or newly-downloaded malware).

To connect the missing causality links, we interpret the HTML and JavaScript content of the pages fetched by the browser and extract all the URLs from the fetched pages. Then, to identify causal edges we look for any URLs that match any of the HTTP fetches that were subsequently visited by the browser. In some cases, URLs contain randomly generated strings, so some requests cannot be matched exactly. In these cases, we apply heuristics based on edit distance to identify the most probable parent of the URL . Finally, for each malware distribution site, we construct its associated distribution network by combining the different malware delivery trees from all landing pages that lead to that site.

Our infrastructure has been live for more than one year, continuously monitoring the web and detecting malicious URLs. In what follows, we report our findings based on analyzing data collected during that time period. Again, recall that we focus here on the pervasiveness of malicious activity (perpetrated by drive-by downloads) that is induced simply by visiting a landing page, thereafter requiring *no* additional interaction on the client's part (*e.g.,* clicking on embedded links). Finally, we note that due to the large scale of our data collection and some infrastructural constraints, a number longitudinal aspects of the web malware problem (*e.g.,* the lifetime of the different malware distribution networks) are beyond the scope of this paper and are a subject of our future investigation.

## 4 Prevalence of Drive-by Downloads

We provide an estimate of the prevalence of web-malware based on data collected over a period of ten months (Jan 2007 - Oct 2007). During that period, we subjected over 60 million URLs for in-depth processing through our verification system. Overall, we detected more than 3 million malicious URLs hosted on more than 180 thousand landing sites. Overall, we observed more than 9 thousand different distribution sites. The findings are summarized in Table 1. Overall, these results show the scope of the problem, but do not necessarily reflect the exposure of end-users to drive-by downloads. In what follows, we attempt to address this question by estimating the overall impact of the malicious web sites.

| | |
|---|---|
| Data collection period | Jan - Oct 2007 |
| Total URLs checked in-depth | 66,534,330 |
| Unique suspicious landing URLs | 3,385,889 |
| Unique malicious landing URLs | 3,417,590 |
| Unique malicious landing sites | 181,699 |
| Unique distribution sites | 9,340 |

Table 1: Summary of collected data.

To study the potential impact of malicious web sites on the end-users, we first examine the fraction of incoming search queries to Google's search engine that return at least one URL labeled as malicious in the results page. Figure 3 provides a running average of this fraction. The graph shows an increasing trend in the search queries that return at least one malicious result, with an average approaching 1.3% of the overall incoming search queries. This finding is troubling as it shows that a significant fraction of search queries return results that may expose the end-user to exploitation attempts.

To further understand the importance of this finding, we inspect the prevalence of malicious sites among the links that appear most often in Google search results. From the top one million URLs appearing in the search engine results, about 6,000 belong to sites that have been verified as malicious at some point during our data collection. Upon closer inspection, we found that these sites

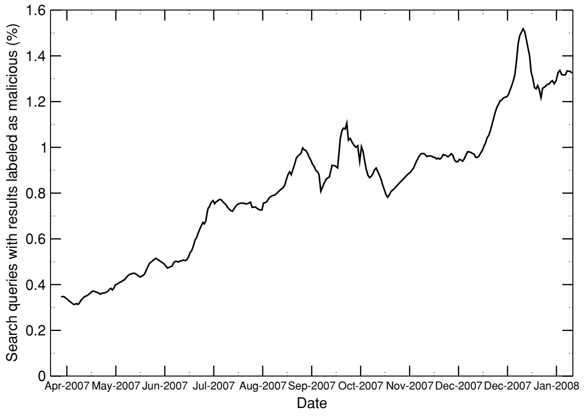


Figure 3: Percentage of search queries that resulted in at least one URL labeled as malicious; 7-day running avg.

appear at uniformly distributed ranks within the top million web sites—with the most popular landing page having a rank of 1,588. These results further highlight the significance of the web malware threat as they show the extent of the malware problem; in essence, about 0.6% of the top million URLs that appeared most frequently in Google's search results led to exposure to malicious activity at some point.

An additional interesting result is the geographic locality of web based malware. Table 2 shows the geographic breakdown of IP addresses of the top 5 malware distribution sites and the landing sites. The results show that a significant number of Chinese-based sites contribute to the drive-by problem. Overall, 67% of the malware distribution sites and 64.6% of the landing sites are hosted in China. These findings provide more evidence [13] of poor security practices by web site administrators, *e.g.*, running out-dated and unpatched versions of the web server software.

| dist. site hosting country | % of all dist. sites | landing site hosting country | % of all landing sites |
|---|---|---|---|
| China | 67.0% | China | 64.4% |
| United States | 15.0% | United States | 15.6% |
| Russia | 4.0% | Russia | 5.6% |
| Malaysia | 2.2% | Korea | 2.0% |
| Korea | 2.0% | Germany | 2.0% |

Table 2: Top 5 Hosting countries

Upon closer inspection of the geographic locality of the web-malware distribution networks as a whole (*i.e.,* the correlation between the location of a distribution site and the landing sites pointing to it), we see that the malware distribution networks are highly localized within common geographical boundaries. This locality varies across different countries, and is most evident in China, with 96% of the landing sites in China pointing to malware distribution servers hosted in that country.

### 4.1 Impact of browsing habits

In order to examine the impact of users' browsing habits on their exposure to exploitation via drive-by downloads, we measure the prevalence of malicious websites across the different website functional categories based on the DMOZ classification [1]. Using a large random sample of about 7.2 million `URLs` , we first map each `URL` to its corresponding DMOZ category. We were able to find the corresponding DMOZ categories for about 50% of these `URLs`[2]. We further inspect each `URL` through our indepth verification system then measure the percentage of malicious `URLs` in each functional category. Figure 4 shows the prevalence of detected malicious and suspicious websites in each top level DMOZ category.

As the graph illustrates, website categories associated with "gray content" (*e.g.,* adult websites) show a stronger connection to malicious content. For instance, about 0.6% of the `URLs` in the Adult category exhibited drive-by download activity upon visiting these websites. These results suggest that users who browse such websites will likely be more exposed to exploitation compared to users who browse websites from the other functional categories. However, an important observation from the same figure is that the distribution of malicious websites is not significantly skewed toward pages that serve gray content. In fact, the distribution shows that malicious websites are generally present in *all* website categories we observed. Overall, these results show that while "safe browsing" habits may limit users' exposure to drive-by downloads it does not provide an effective safeguard against exploitation.

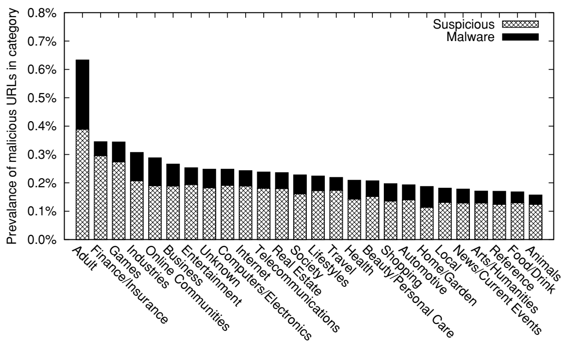


Figure 4: Prevalence of suspicious and malicious pages.

## 5 Malicious Content Injection

In Section 4, we showed that exposure to web-malware is not strongly tied to a particular browsing habit. Our assertion is that this is due, in part, to the fact that drive-by downloads are triggered by visiting staging sites that are not necessarily of malicious intent but have content that lures the visitor into the malware distribution network.

In this section, we validate this conjecture by studying the properties of the web sites that participate in the malware delivery trees. As discussed in Section 2, attackers use a number of techniques to control the content of benign web sites and turn them into nodes in the malware distribution networks. These techniques can be divided into two categories: web server compromise and third party contributed content (*e.g.,* blog posts). Unfortunately, it is generally difficult to determine the exact contribution of either category. In fact, in some cases even manual inspection of the content of each web site may not lead to conclusive evidence regarding the manner in which the malicious content was injected into the web site. Therefore, in this section we provide insights into some features of these web sites that may explain their presence in the malware delivery trees. We only focus on the features that we can determine in an automated fashion. Specifically, where possible, we first inspect the version of the software running on the web server for each landing site. Additionally, we explore one important angle that we discovered which contributes significantly to the distribution of web malware—namely, drive-by downloads via Ads.

### 5.1 Web Server Software

We first begin by examining (where possible) the software running on the web-servers for all the landing sites that lead to the malware distribution sites. Specifically, we collected all the "`Server`" and "`X-Powered-By`" header tokens from each landing page (see Table 3). Not surprisingly, of those servers that reported this information, a significant fraction were running outdated versions of software with well known vulnerabilities[3]. For example, 38.1% of the Apache servers and 39.9% of servers with PHP scripting support reported a version with security vulnerabilities. Overall, these results reflect the weak security practices applied by the web site administrators. Clearly, running unpatched software with known vulnerabilities increases the risk of content control via server exploitation.

| Srv. Software | count | Unknown | Up-to-date | Old |
|---|---|---|---|---|
| Apache | 55,088 | 26.5% | 35.5% | 38% |
| Microsoft IIS | 113,905 | n/a | n/a | n/a |
| Unknown | 12,706 | n/a | n/a | n/a |
| Scripting | | | | |
| PHP | 27,873 | 8.5% | 51.6% | 39.9% |

Table 3: Server version for landing sites. In the case of Microsoft IIS, we could not verify their version.

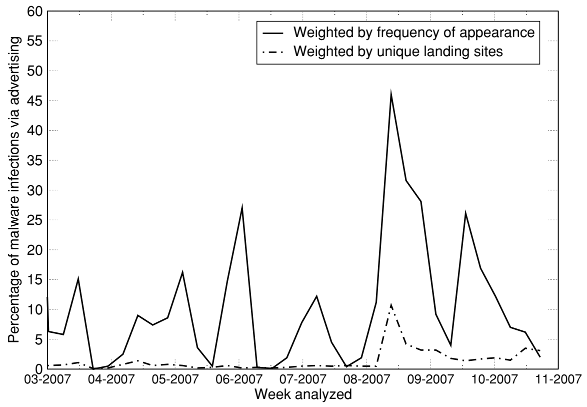


Figure 5: Percentage of landing sites potentially infecting visitors via malicious advertisements, and their relative share in the search results.

## 5.2 Drive-by Downloads via Ads

Today, the majority of Web advertisements are distributed in the form of third party content to the advertising web site. This practice is somewhat worrisome, as a web page is only as secure as it's weakest component. In particular, even if the web page itself does not contain any exploits, insecure Ad content poses a risk to advertising web sites. With the increasing use of Ad syndication (which allows an advertiser to sell advertising space to other advertising companies that in turn can yet again syndicate their content to other parties), the chances that insecure content gets inserted somewhere along the chain quickly escalates. Far too often, this can lead to web pages running advertisements to untrusted content. This, in itself, represents an attractive avenue for distributing malware, as it provides the adversary with a way to inject content to web sites with large visitor base without having to compromise any web server.

To assess the extent of this behavior, we estimate the overall contribution of Ads to drive-by downloads. To do so, we construct the malware delivery trees from all detected malicious URLs following the methodology described in Section 3. For each tree, we examine every intermediary node for membership in a set of $2,000$ well known advertising networks. If any of the nodes qualify, we count the landing site as being infectious via Ads. Moreover, to highlight the impact of the malware delivered via Ads relative to the other mechanisms, we weight the landing sites associated with Ads based on the frequency of their appearance in Google search results compared to that of all landing sites. Figure 5 shows the percentage of landing sites belonging to Ad networks. On average, $2\%$ of the landing sites were delivering malware via advertisements. More importantly, the overall weighted share for those sites was substantial—on average, $12\%$ of the overall search results that returned landing pages were associated with malicious content due to unsafe Ads. This result can be explained by the fact that Ads normally target popular web sites, and so have a much wider reach. Consequently, even a small fraction of malicious Ads can have a major impact (compared to the other delivery mechanisms).

Another interesting aspect of the results shown in Figure 5 is that Ad-delivered drive-by downloads seem to appear in sudden short-lived spikes. This is likely due to the fact that Ads appearing on several advertising web sites are centrally controlled, and therefore allow the malicious content to appear on thousands of web sites sites almost instantaneously. Similarity, once detected, these Ads are removed simultaneously, and so disappear as quickly as they appeared. For this reason, we notice that drive-by downloads delivered by other content injection techniques (*e.g.,* individual web servers compromise) have more lasting effect compared to Ad delivered malware, as each web site must be secured independently.

The general practice of Ad syndication contributes significantly to the rise of Ad delivered malware. Our results show that overall 75% of the landing sites that delivered malware via Ads use multiple levels of Ad syndication. To understand how far trust would have to extend in order to limit the Ad delivered drive-by down-

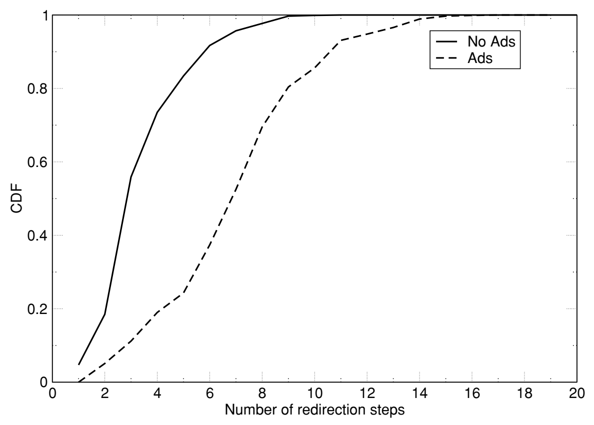


Figure 6: CDF of the number of redirection steps for Ads that successfully delivered malware.

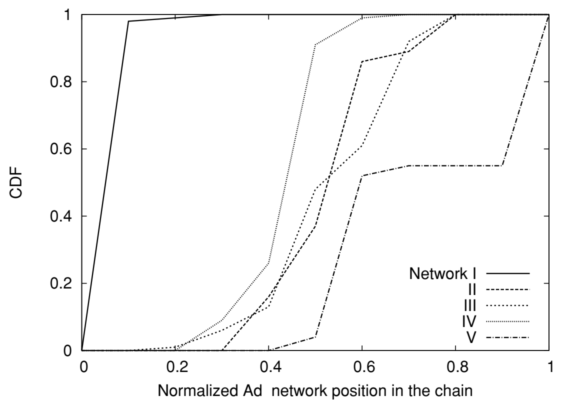


Figure 7: CDF of the normalized position of the top five Ad networks most frequently participating in malware delivery chains.

loads, we plot the distribution of the path length from the landing site leading to the malware distribution sites for each delivery tree. The edges connecting the nodes in these paths reflect the number of redirects a browser has to follow before receiving the final payload. Hence, for syndicated Ads that delivered malware the path length is indicative of the number of syndication steps before reaching the final Ad; in our case, the malware payload. Figure 6 shows the distribution of the number of redirects for syndicated Ads that delivered malware relative to the other malicious landing `URLs`. The results are quite telling: malware delivered via Ads exhibits longer delivery chains, in $50\%$ percent of all cases, more than 6 redirection steps were required before receiving the malware payload. Clearly, it is increasingly difficult to maintain trust along such long delivery chains.

Inspecting the delivery trees that featured syndication reveals a total of 55 unique Ad networks participating in these trees. We further studied the relative role of the different networks by evaluating the frequency of appearance of each Ad network in the malware delivery trees. Interestingly, our results show that five advertising networks appear in approximately $75\%$ of all malware delivery trees. Figure 7 shows the distribution of the relative position of each network in the malware delivery chains it participated in. The normalized position is calculated by dividing the index of the Ad network in each chain by the length of the chain. The graph shows that these advertising networks split into three different categories: In the first category, which includes network `I`, the advertising network appears at the beginning of the delivery chain. In the second category, which includes networks `II-IV`, advertising networks appear frequently in the middle of the delivery chains. In both these categories advertising networks do not participate directly in delivering malware. However, the relative position of networks in the delivery chain may be used as an indication of their relationship with the malware distribution sites – the deeper a network's relative position the closer it is related to the malware distribution site. Finally, in the third category, indicated by network `V`, our analysis revealed that in almost $50\%$ of all incidents, the advertising network is directly delivering malware. For example, advertising network `V` pushes Ads that install malware in the form of a browser toolbar.

Finally we further elucidate this problem via an interesting example from our data corpus. The landing page in our example refers to a Dutch radio station's web site. The radio station in question was showing a banner advertisement from a German advertising site. Using JavaScript, that advertiser redirected to a prominent advertiser in the US, which in turn redirected to yet another advertiser in the Netherlands. That advertiser redirected to another advertisement (also in the Netherlands) that contained obfuscated JavaScript, which when unobfuscated, pointed to yet another JavaScript hosted in Austria. The final JavaScript was encrypted and redirected the browser via multiple `IFRAME`s to *adxtnet.net*, an exploit site hosted in Austria. This resulted in the automatic installation of multiple Trojan Downloaders. While it is unlikely that the initial advertising companies were aware of the malware installations, each redirection gave another party control over the content on the original web page—with predictable consequences.

## 6 Malware Distribution Infrastructure

In this section, we explore various properties of the hosting infrastructure for web malware. In particular, we explore the size of of the malware distribution networks, and examine the distribution of binaries hosted across sites. We argue that such analysis is important, as it sheds light on the sophistication of the hosting infrastructures and the level of malfeasance we see today. As is the case with other recent malware studies (e.g., [5, 26, 21]) we hope that this analysis will be of benefit to researchers and practitioners alike.

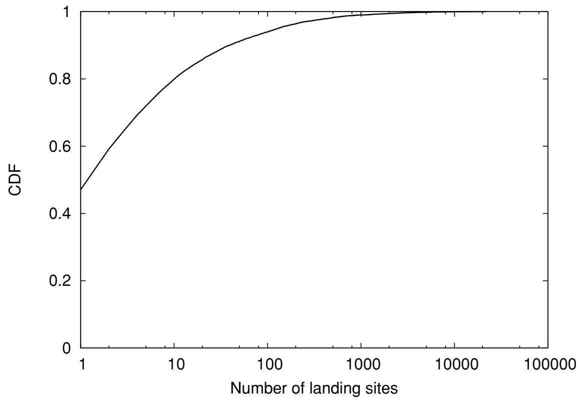


Figure 8: CDF of the number of landing sites pointing to a particular malware distribution site.

For the remaining discussion, recall that a malware distribution network constitutes all the landing sites that point to a single distribution site. Using the methodology described in Section 3, we identified the distribution networks associated with each malware distribution site. We first evaluate their size in terms of the total number of landing sites that point to them. Figure 8 shows the distribution of sizes for the different distribution networks.

The graph reveals two main types of malware distribution networks: (1) networks that use only one landing site, and (2) networks that have multiple landing sites. As the graph shows, distribution networks can grow to have well over 21,000 landing sites pointing to them. That said, roughly 45% of the detected malware distribution sites used only a single landing site at a time. We manually inspected some of these distribution sites and found that the vast majority were either subdomains on free hosting services, or short-lived domains that were created in large numbers. It is likely, though not confirmed, that each of these sites used only a single landing site as a way to slip under the radar and avoid detection.

Next, we examine the network location of the malware

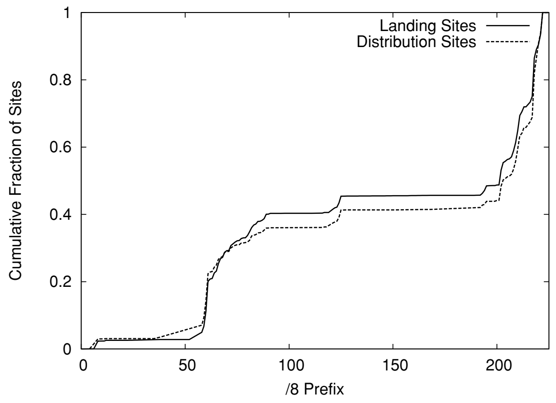


Figure 9: The cumulative fraction of malware distribution sites over the $/8$ IP prefix space.

distribution servers and the landing sites linking to them. Figure 9 shows that the malware distribution sites are concentrated in a limited number of /8 prefixes. About 70% of the malware distribution sites have IP addresses within `58.* -- 61.*` and `209.* -- 221.*` network ranges. Interestingly, Anderson *et al.* [5] observed comparable IP space concentrations for the scam hosting infrastructure. The landing sites, however exhibit relatively more IP space diversity; Roughly 50% of the landing sites fell in the above ranges.

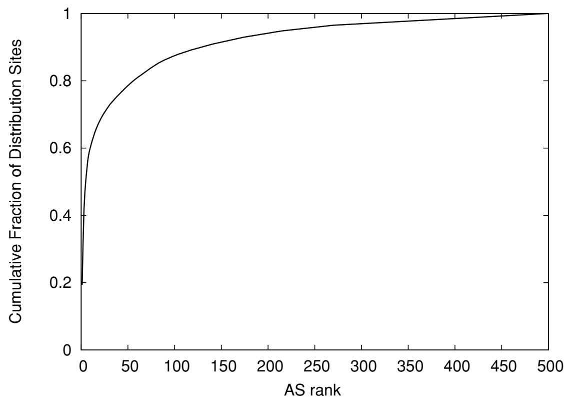


Figure 10: The cumulative fraction of the malware distribution sites across the different ASes.

We further investigated the Autonomous System (AS) locality of the malware distribution sites by mapping their IP addresses to the AS responsible for the longest matching prefixes for these IP addresses. We use the latest BGP snapshot from Routeviews [23] to do the IP to AS mapping. Our results show that all the malware distribution sites' IP addresses fall into a relatively small set of ASes — only 500 as of this writing. Figure 10 shows the cumulative fraction of these sites across the ASes hosting them (sorted in descending order by the number of sites in each AS). The graph further shows the highly nonuniform concentration of the malware distribution sites: $95\%$ of these sites map to only $210$ ASes. Finally, the results of mapping the landing sites (not shown) produced $2,517$ ASes with $95\%$ of the sites falling in these $500$ ASes.

Lastly, the distribution of malware across domains also gives rise to some interesting insights. Figure 11 shows the distribution of the number of unique malware binaries (as inferred from MD5 hashes) downloaded from each malware distribution site. As the graph shows, approximately 42% of the distribution sites delivered a single malware binary. The remaining distribution sites hosted multiple distinct binaries over their observation period in our data, with 3% of the servers hosting more than 100 binaries. In many cases, we observed that the multiple payloads reflect deliberate obfuscation attempts to evade detection. In what follows, we take a more in-depth look by studying the different forms of relationships among the various distribution networks.

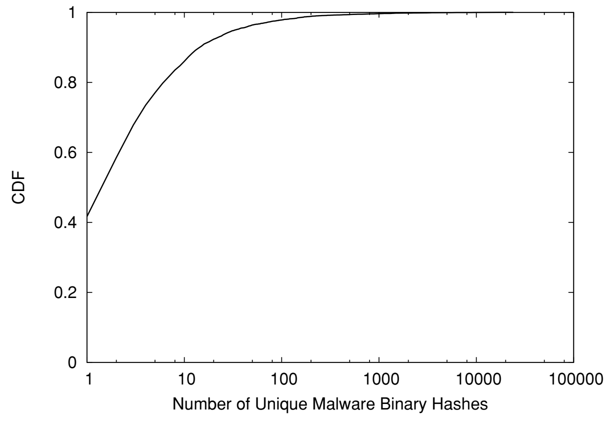


Figure 11: CDF of the number of unique binaries downloaded from each malware distribution site.

## 6.1 Relationships Among Networks

To gain a better perspective on the degree of connectivity between the distribution networks, we investigate the common properties of the hosting infrastructure across the malware distribution sites. We also evaluate the degree of overlap among the landing sites linking to the different malware distribution sites.

**Malware hosting infrastructure.** Throughout our measurement period we detected $9,430$ malware distribution sites. In $90\%$ of the cases each site is hosted on a single IP address. The remaining $10\%$ sites are hosted on IP addresses that host multiple malware distribution sites. Our results show IP addresses that hosted up to $210$ malware distribution sites. Closer inspection revealed that these addresses refer to public hosting servers that allow users to create their own accounts. These accounts appear as sub-folders of the the virtual hosting server DNS name (*e.g.,* `512j.com/akgy`, `512j.com/alavin`, `512j.com/anti`) or in many cases as separate DNS aliases that resolve to the IP address of the hosting server. We also observed several cases where the hosting server is a public blog that allows users to have their own pages (*e.g.,* `mihanblog.com/abadan2`, `mihanblog.com/askbox`).

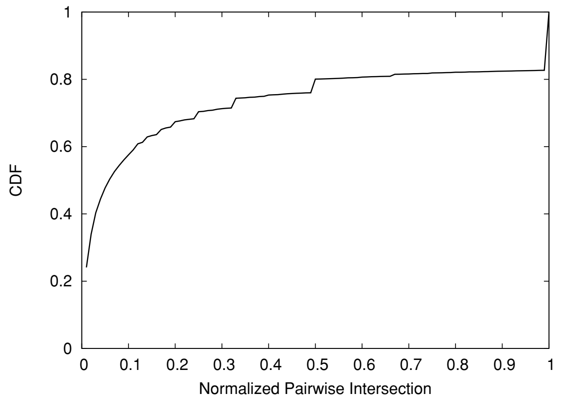


Figure 12: CDF of the normalized pairwise intersection between landing sites across distribution networks.

**Overlapping landing sites.** We further evaluate the overlap between the landing sites that point to the different malware distribution sites. To do so, we calculate the pairwise intersection between the sets of the landing sites pointing to each of the distribution sites in our data set. For a distribution network $i$ with a set of landing sites $X_i$ and network $j$ with the set of landing sites $X_j$, the normalized pairwise intersection of the two networks, $C_{i,j}$, is calculated as,

$$C_{i,j} = \frac{|X_i \cap X_j|}{|X_i|} \qquad (1)$$

Where $|X|$ is the number of elements in the set $X$. Interestingly, our results showed that $80\%$ of the distribution networks share at least one landing page. Figure 12 shows the normalized pair-wise landing sets intersection across these distribution networks. The graph reveals a strong overlap among the landing sites for the related network pairs. These results suggest that many landing sites are shared among multiple distribution networks. For example, in several cases we observed landing pages with multiple `IFRAME`s linking to different malware distribution sites. Finally, we note that the sudden jump to a pair-wise score of one is mostly due to network pairs in which the landing sites for one network are a subset of those for the other network.

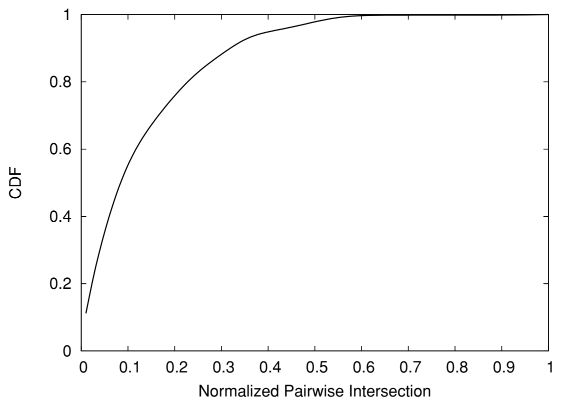


Figure 13: CDF of the normalized pairwise intersection between malware hashes across distribution networks.

**Content replication across malware distribution sites.** We finally evaluate the extent to which malware is replicated across the different distribution sites. To do so, we use the same metric in Equation 1 to calculate the normalized pairwise intersection of the set of malware hashes served by each pair of distribution sites. Our results show that in $25\%$ of the malware distribution sites, at least one binary is shared between a pair of sites. While malware hashes exhibit frequent changes as a result of obfuscation, our results suggest that there is still a level of content replication across the different sites. Figure 13 shows the normalized pair-wise intersection of the malware sets across these distribution networks. As the graph shows, binaries are less frequently shared between distribution sites compared to landing sites, but taken as a whole, there is still a non-trivial degree of similarity among these networks.

## 7 Post Infection Impact

Recall that upon visiting a malicious `URL`, the browser downloads the initial exploit. The exploit (in most cases, `javascript`) targets a vulnerability in the browser or one of its plugins and takes control of the infected system, after which it retrieves and runs the malware executable(s) downloaded from the malware distribution site. Rather than inspecting the behavior of each phase in isolation, our goal is to give an overview of the collective changes that happen to the system state after visiting a malicious `URL` . Figure 14 shows the distribution of the number of Windows executables downloaded after visiting a malicious `URL` as observed from monitoring the interaction between the browser and the malware distribution site. As the graph shows, visiting malicious `URLs` can lead to a large number of downloads (8 on average, but as large as 60 in the extreme case).

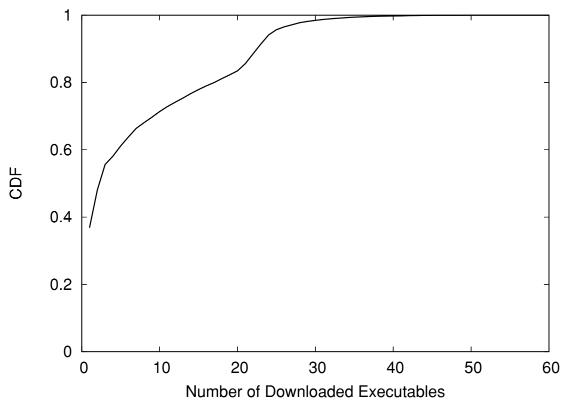


Figure 14: CDF of the number of downloaded executables as a result of visiting a malicious `URL`

Another noticeable outcome is the increase in the number of running processes on the virtual machine. This increase is associated with the automatic execution of binaries. For each landing `URL` , we collected the number of processes that were started on the guest operating system after being infected with malware. Figure 15 shows the CDF of the number of processes launched after the system is infected. As the graph shows visiting malicious `URLs` produces a noticeable increase in the number of processes, in some cases, inducing so much overhead that they "crashed" the virtual machine.

Additionally, we examine the type of registry changes that occur when the malware executes. Overall, we detected registry changes after visiting $57.5\%$ of the landing pages. We divide these changes into the following categories: *BHO* indicates that the malware installed a Browser Helper Object that can access privileged state in the browser; *Preferences* means that the browser home page, default search engine or name server where changed by the malware; *Security* indicates that

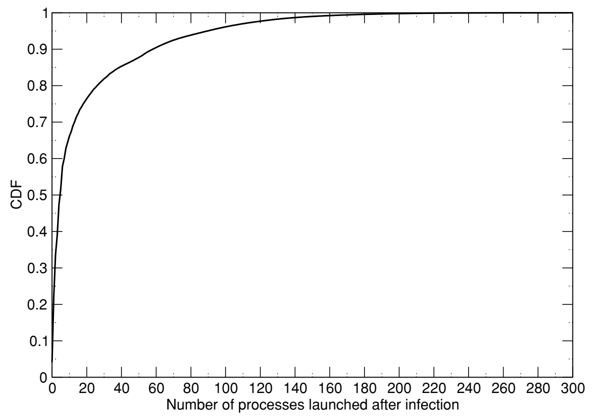


Figure 15: CDF of the number of processes started after visiting a malicious URL

malware changed firewall settings or even disabled automatic software updates; *Startup* indicates that the malware is trying to persist across reboots. Notice that these categories are not mutually exclusive (*i.e.,* a single malicious URL may cause changes in multiple categories). Table 4 summarizes the percentage of registry changes per category. Notice that "Startup" changes are more prevalent indicating that malware tries to persist even after the machine is rebooted.

| Category | BHO | Preferences | Security | Startup |
|---|---|---|---|---|
| URLs % | 6.99% | 23.5% | 36.18% | 51.27% |

Table 4: Registry changes from drive-by downloads.

In addition to the registry changes, we analyzed the network activity of the virtual machine post infection. In our system, the virtual machines are allowed to perform only DNS and HTTP connections. Table 5 shows the percentage of connection attempts per destination port. Even though we omit the HTTP connections originating from the browser, HTTP is still the most prevalent port for malicious activity post-infection. This is due to "downloader" binaries that fetch, in some cases, up to 60 binaries over HTTP. We also observe a significant percentage of connection attempts to typical IRC ports, accounting for more than 50% of all non-HTTP connections. As a number of earlier studies have already shown (e.g., [6, 19, 8, 21, 22, 12]), the IRC connection attempts are most likely for unwillingly (to the owner) adding the compromised machine to an IRC botnet, confirming the earlier conjecture by Provos *et al.* [20] regarding the connection between web malware and botnets. More detailed examples of malware's behavior can be found in Polychronakis *et al.* [18].

| Protocol/Port | Connections % |
|---|---|
| HTTP (80, 8080) | 87% |
| IRC (6660-7001) | 8.3% |
| FTP (21) | 0.9% |
| UPnP (1900) | 0.8% |
| Mail (25) | 0.75% |
| Other | 2.25% |

Table 5: Most frequently contacted ports directly by the downloaded malware.

## 7.1 Anti-virus engine detection rates

As we discussed earlier, web based malware uses a *pull-based* delivery mechanism in which a victim is required to visit the malware hosting server or any URL linking to it in order to download the malware. This behavior puts forward a number of challenges to defense mechanisms (*e.g.,* malware signature generation schemes) mainly due to the inadequate coverage of the malware collection system. For example, unlike active scanning malware which uses a *push-based* delivery mechanism (and so sufficient placement of honeypot sensors can provide good coverage), the web is significantly more sparse and, therefore, more difficult to cover.

In what follows, we evaluate the potential implications of the web malware delivery mechanism by measuring the detection rates of several well known anti-virus engines. Specifically, we evaluate the detection rate of each anti-virus engine against the set of *suspected* malware samples collected by our infrastructure. Since we can not rely on anti-virus engines, we developed a heuristic to detect these suspected binaries before subjecting them to the anti-virus scanners. For each inspected URL via our in-depth verification system we test whether visiting the URL caused the creation of at least one new process on the virtual machine. For the URLs that satisfy this condition, we simply extract any binary[4] download(s) from the recorded HTTP response and "flag" them as suspicious.

We applied the above methodology to identify suspicious binaries on a daily basis over a one month period of April, 2007. We subject each binary for each of the anti-virus scanners using the latest virus definitions on that day. Then, for an anti-virus engine, the detection rate is simply the number of detected (flagged) samples divided by the total number of suspicious malware instances inspected on that day. Figure 16 illustrates the individual detection rates of each of the anti-virus engines. The graph reveals that the detection capability of the anti-virus engines is lacking, with an average detection rate of 70% for the best engine. These results are disturbing as they show that even the best anti-virus engines in the market (armed with their latest definitions) fail to cover a significant fraction of web malware.

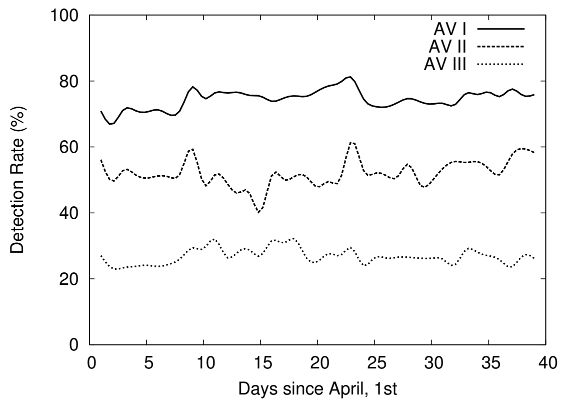


Figure 16: Detection rates of 3 anti-virus engines.

**False Positives.** Notice that the above strategy may falsely classify benign binaries as malicious. To evaluate the false positives, we use the following heuristic: we optimistically assume that all suspicious binaries will eventually be discovered by the anti-virus vendors. Using the set of suspicious binaries collected over a month historic period, we re-scan all undetected binaries two months later (in July, 2007) using the latest virus definitions. Then, all undetected binaries from the rescanning step are considered false positives. Overall, our results show that the earlier analysis is fairly accurate with false positive rates of less than 10%. We further investigated a number of binaries identified as false positives and found that a number of popular installers exhibit a behavior similar to that of drive-by downloads, where the installer process first runs and then downloads the associated software package. To minimize the impact of false positives, we created a white-list of all known benign downloads, and all binaries in the white-list are exempted from the analysis in this paper.

Of course, we are being overly conservative here as our heuristic does not account for binaries that are never detected by any anti-virus engine. However, for our goals, this method produces an upper bound for the resulting false positives. As an additional benchmark we asked for direct feedback from anti-virus vendors about the accuracy of the undetected binaries that we (now) share with them. On average, they reported about 6% false positives in the shared binaries, which is within the bounds of our prediction.

## 8 Discussion

Undoubtedly, the level of malfeasance on the Internet is a cause for concern. That said, while our work to date has shown that the prevalence of web-malware is indeed a serious threat, the analysis herein says nothing about the number of visitors that become infected as a result of visiting a malicious page. In particular, we note that since our goal is to survey the landscape, our infrastructure is intentionally configured to be vulnerable to a wide range of attacks; hopefully, savvy computer users who diligently apply software updates would be far less vulnerable to infection. To be clear, while our analysis unequivocally shows that millions of users are exposed to malicious content every day, without a wide-scale browser vulnerability study, the actual number of compromises remains unknown. Nonetheless, we believe the pervasive nature of the results in this study elucidates the state of the malware problem today, and hopefully, serves to educate both users, web masters and other researchers about the security challenges ahead.

Lastly, we note that several outlets exists for taking advantage of the results of our infrastructure. For instance, the data that Google uses to flag search results is freely available through the Safe Browsing API [2], as well as via the Safe Browsing diagnostic page [3]. We hope these services prove to be of benefit to the greater community at large.

## 9 Related Work

Virtual machines have been used as honeypots for detecting unknown attacks by several researchers [4, 16, 17, 25, 26]. Although, honeypots have traditionally been used mostly for detecting attacks against servers, the same principles also apply to client honeypots (e.g., an instrumented browser running on a virtual machine). For example, Moshchuk *et al.* used client-side techniques to study spyware on the web (by crawling 18 million `URLs` in May 2005 [17]). Their primary focus was not on detecting drive-by downloads, but in finding links to executables labeled spyware by an adware scanner. Additionally, they sampled $45,000$ `URLs` for drive-by downloads and showed a *decrease* over time. However, the fundamental limitation of analyzing the malicious nature of `URLs` discovered by "spidering" is that a crawl can only follow content links, whereas the malicious nature of a page is often determined by the web hosting infrastructure. As such, while the study of Moshchuk *et al.* provides valuable insights, a truly comprehensive analysis of this problem requires a much more in-depth crawl of the web. As we were able to analyze many billions of `URLs` , we believe our findings are more representative of the state of the overall problem.

More closely related is the work of Provos *et al.* [20] and Seifert *et al.* [24] which raised awareness of the threat posed by drive-by downloads. These works are aimed at explaining how different web page components are used to exploit web browsers, and provides an overview of the different exploitation techniques in use today. Wang *et al.* proposed an approach for detecting exploits against Windows XP when visiting webpages in Internet Explorer [26]. Their approach is capable of detecting zero-day exploits against Windows and can determine which vulnerability is being exploited by exposing Windows systems with different patch levels to dangerous `URLs`. Their results, on roughly $17,000$ `URLs`, showed that about 200 of these were dangerous to users.

This paper differs from all of these works in that it offers a far more comprehensive analysis of the different aspects of the problem posed by web-based malware, including an examination of its prevalence, the structure of the distribution networks, and the major driving forces.

Lastly, malware detection via dynamic tainting analysis may provide deeper insight into the mechanisms by which malware installs itself and how it operates [10, 15, 27]. In this work, we are more interested in structural properties of the distribution sites themselves, and how malware behaves once it has been implanted. Therefore, we do not employ tainting because of its computational expense, and instead, simply collect changes made by the malware that do not require having the ability to trace the information flow in detail.

## 10 Conclusion

The fact that malicious `URLs` that initiate drive-by downloads are spread far and wide raises concerns regarding the safety of browsing the Web. However, to date, little is known about the specifics of this increasingly common malware distribution technique. In this work, we attempt to fill in the gaps about this growing phenomenon by providing a comprehensive look at the problem from several perspectives. Our study uses a large scale data collection infrastructure that continuously detects and monitors the behavior of websites that perpetrate drive-by downloads. Our in-depth analysis of over 66 million `URLs` (spanning a 10 month period) reveals that the scope of the problem is significant. For instance, we find that $1.3\%$ of the incoming search queries to Google's search engine return at least one link to a malicious site.

Moreover, our analysis reveals several forms of relations between some distribution sites and networks. A more troubling concern is the extent to which users may be lured into the malware distribution networks by content served through online Ads. For the most part, the syndication relations that implicitly exist in advertising networks are being abused to deliver malware through Ads. Lastly, we show that merely avoiding the dark corners of the Internet does not limit exposure to malware. Unfortunately, we also find that even state-of-the-art anti-virus engines are lacking in their ability to protect against drive-by downloads. While this is to be expected, it does call for more elaborate defense mechanisms to curtail this rapidly increasing threat.

## Acknowledgments

We would like to thank Oliver Fisher, Dean McNamee, Mark Palatucci and Ke Wang for their help with Google's malware detection infrastructure. This work was funded in part by NSF grants CNS-0627611 and CNS-0430338.

## References

[1] The open directory project. See `http://www.news.com/2100-1023-877568.html`.

[2] Safe Browsing API, June 2007. See `http://code.google.com/apis/safebrowsing/`.

[3] Safe Browsing diagnostic page, May 2008. See `http://www.google.com/safebrowsing/diagnostic?site=yoursite.com`.

[4] ANAGNOSTAKIS, K. G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., AND KEROMYTIS, A. D. Detecting Targeted Attacks Using Shadow Honeypots.

[5] ANDERSON, D. S., FLEIZACH, C., SAVAGE, S., AND VOELKER, G. M. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *Proceedings of the USENIX Security Symposium* (August 2007).

[6] BARFORD, P., AND YAGNESWARAN, V. *An Inside Look at Botnets*. Advances in Information Security. Springer, 2007.

[7] BEM, J., HARIK, G., LEVENBERG, J., SHAZEER, N., AND TONG, S. Large scale machine learning and methods. US Patent: 7222127.

[8] COOKE, E., JAHANIAN, F., AND MCPHERSON, D. The Zombie Roundup: Understanding, Detecting, and Disturbing Botnets. In *Proceedings of the first Workshop on*

*Steps to Reducing Unwanted Traffic on the Internet* (July 2005).

[9] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation* (Dec 2004), pp. 137–150.

[10] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference* (June 2007).

[11] FRANKLIN, J., PAXSON, V., PERRIG, A., AND SAVAGE, S. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (October 2007).

[12] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting Malware Infection through IDS-driven Dialog Correlation. In *Proceedings of the $16^th$ USENIX Security Symposium* (2007), pp. 167–182.

[13] MODADUGU, N. Web Server Software and Malware, June 2007. See `http://googleonlinesecurity.blogspot.com/2007/06/web-server-software-and-malware.html`.

[14] MOORE, D., VOELKER, G. M., AND SAVAGE, S. Inferring Internet Denial of Service Activity. In *Proceedings of $10^{th}$ USENIX Security Symposium* (Aug. 2001).

[15] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007).

[16] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S., AND LEVY, H. SpyProxy: Execution-based Detection of Malicious Web Content.

[17] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S., AND LEVY, H. A crawler-based study of spyware in the web. In *Proceedings of Network and Distributed Systems Security Symposium* (2006).

[18] POLYCHRONAKIS, M., MAVROMMATIS, P., AND PROVOS, N. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (April 2008).

[19] PROJECT, H., AND ALLIANCE, R. Know your enemy: Tracking Botnets, March 2005. See `http://www.honeynet.org/papers/bots/`.

[20] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the first USENIX workshop on hot topics in Botnets (HotBots'07).* (April 2007).

[21] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference (IMC)* (Oct., 2006), pp. 41–52.

[22] RAMACHANDRAN, A., FEAMSTER, N., AND DAGON, D. Revealing Botnet Membership using DNSBL Counter-Intelligence. In *Proceedings of the $2^{nd}$ Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)* (July 2006).

[23] The Route Views Project. `http://www.antc.uoregon.edu/route-views/`.

[24] SEIFERT, C., STEENSON, R., HOLZ, T., BING, Y., AND DAVIS, M. A. Know Your Enemy: Malicious Web Servers. `http://www.honeynet.org/papers/mws/`, August 2007.

[25] WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated web patrol with strider honeymonkeys. In *Proceedings of Network and Distributed Systems Security Symposium* (2006), pp. 35–49.

[26] WANG, Y.-M., NIU, Y., CHEN, H., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Strider honeymonkeys: Active, client-side honeypots for finding malicious websites. See `http://research.microsoft.com/users/shuochen/HM.PDF`.

[27] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference of Computer and Communication Security* (October 2007).

## Notes

[1]Some compromised web servers also trigger dialog windows asking users to manually download and run malware. However, this analysis considers only malware installs that require no user interaction.

[2]This mapping is readily available at Google.

[3]We consider a version as outdated if it is older than the latest corresponding version released by January, 2007 (the start date for our data collection).

[4]We restrict our analysis to Windows executables identified by searching for PE headers in each payload.

# Securing Frame Communication in Browsers

Adam Barth
*Stanford University*
*abarth@cs.stanford.edu*

Collin Jackson
*Stanford University*
*collinj@cs.stanford.edu*

John C. Mitchell
*Stanford University*
*mitchell@cs.stanford.edu*

## Abstract

Many web sites embed third-party content in frames, relying on the browser's security policy to protect them from malicious content. Frames, however, are often insufficient isolation primitives because most browsers let framed content manipulate other frames through navigation. We evaluate existing frame navigation policies and advocate a stricter policy, which we deploy in the open-source browsers. In addition to preventing undesirable interactions, the browser's strict isolation policy also hinders communication between cooperating frames. We analyze two techniques for inter-frame communication. The first method, fragment identifier messaging, provides confidentiality without authentication, which we repair using concepts from a well-known network protocol. The second method, `postMessage`, provides authentication, but we discover an attack that breaches confidentiality. We modify the `postMessage` API to provide confidentiality and see our modifications standardized and adopted in browser implementations.

## 1 Introduction

Web sites contain content from sources of varying trustworthiness. For example, many web sites contain third-party advertising supplied by advertisement networks or their sub-syndicates [6]. Other common aggregations of third-party content include Flickr albums [12], Facebook badges [9], and personalized home pages offered by the three major web portals [15, 40, 28]. More advanced uses of third-party components include Yelp's use of Google Maps [14] to display restaurant locations and the Windows Live Contacts gadget [27]. A web site combining content from multiple sources is called a *mashup*, with the party combining the content called the *integrator* and integrated content called a *gadget*. In simple mashups, the integrator does not intend to communicate with the gadgets and requires only that the browser isolate frames. In more complex mashups, the integrator does intend to communicate with the gadgets and requires secure inter-frame communication.

In this paper, we study the contemporary web version of a recurring problem in computer systems: isolating untrusted, or partially trusted, software components while providing secure inter-component communication. Whenever a site integrates third-party content, such as an advertisement, a map, or a photo album, the site runs the risk of incorporating malicious content. Without isolation, malicious content can compromise the confidentiality and integrity of the user's session with the integrator. While the browser's well-known "same-origin policy" [34] restricts script running in one frame from manipulating content in another frame, the browser uses a different policy to determine whether one frame is allowed to navigate (change the location of) another frame. Although restricting navigation is essential to providing isolation, navigation also enables one form of inter-frame communication used in mashup frameworks from leading companies. Furthermore, we show that an attacker can use frame navigation to attack another inter-frame communication mechanism, `postMessage`.

**Isolation.** We examine the browser frame as an isolation primitive. Because frames can contain untrusted content, the browser's security policy restricts frame interactions. Many browsers, however, insufficiently restrict the ability of one frame to navigate another frame to a new location. These overly permissive frame navigation policies lead to a variety of attacks, which we demonstrate against the Google AdSense login page and the iGoogle gadget aggregator. To prevent these attacks, we propose tightening the browser's frame navigation policy while maintaining compatibility with existing web content. We have collaborated with browser vendors to deploy this policy in Firefox 3 and Safari 3.1. As the policy is already implemented in Internet Explorer 7, the policy is now deployed in the three most-used browsers.

|  | Confidentiality | Authentication | Network Analogue |
|---|---|---|---|
| Fragment identifier channel | ✓ |  | Public Key Encryption |
| `postMessage` channel |  | ✓ | Public Key Signatures |
| `postMessage` (our proposal) | ✓ | ✓ | SSL/TLS |

Table 1: Security properties of frame communication channels

**Communication.** With strong isolation, frames are limited in their interactions, raising the issue of how isolated frames can cooperate as part of a mashup. We analyze two techniques for inter-frame communication: fragment identifier messaging and `postMessage`. The results of our analysis are summarized in Table 1.

- *Fragment identifier messaging* uses characteristics of frame navigation to send messages between frames. As it was not designed for communication, the channel has less-than-desirable security properties: messages are confidential but senders are not authenticated. To understand these properties, we draw an analogy between this communication channel and a network channel in which senders encrypt their messages to their recipient's public key. For concreteness, we examine the `Microsoft.Live.Channels` library [27], which uses fragment identifier messaging to let the Windows Live Contacts gadget communicate with its integrator. The protocol used by Windows Live is analogous to the Needham-Schroeder public-key protocol [29]. We discover an attack on this protocol, related to Lowe's anomaly in the Needham-Schroeder protocol [23], in which a malicious gadget can impersonate the integrator to the Contacts gadget. We suggested a solution based on Lowe's improvement to the Needham-Schroeder protocol [23], and Microsoft implemented and deployed our suggestion within days.

- *postMessage* is a new browser API designed for inter-frame communication [19]. `postMessage` is implemented in Opera, Internet Explorer 8, Firefox 3, and Safari. Although `postMessage` has been deployed since 2005, we demonstrate an attack on the channel's confidentiality using frame navigation. In light of this attack, the `postMessage` channel provides authentication but lacks confidentiality, analogous to a channel in which senders cryptographically sign their messages. To secure the channel, we propose a change to the `postMessage` API. We implemented our change in patches for Safari and Firefox. Our proposal has been adopted by the HTML 5 working group, Internet Explorer 8, Firefox 3, and Safari.

**Organization.** The remainder of the paper is organized as follows. Section 2 details the threat model for these attacks. Section 3 surveys existing frame navigation policies and converges browsers on a secure policy. Section 4 analyzes two frame communication mechanisms, demonstrates attacks, and proposes defenses. Section 5 describes related work. Section 6 concludes.

## 2 Threat Model

In this paper, we are concerned with securing in-browser interactions from malicious attackers. We assume an honest user employs a standard web browser to view content from an honest web site. A malicious "web attacker" attempts to disrupt this interaction or steal sensitive information. Typically, a web attacker places malicious content (e.g., JavaScript) in the user's browser and modifies the state of the browser, interfering with the honest session. To study the browser's security policy, which determines the privileges of the attacker's content, we define the web attacker threat model below.

**Web Attacker.** A *web attacker* is a malicious principal who owns one or more machines on the network. In order to study the security of browsers when rendering malicious content, we assume that the browser gets and renders content from the attacker's web site.

- **Network Abilities.** The web attacker has no special network abilities. In particular, the web attacker can send and receive network messages only from machines under his or her control, possibly acting as a client or server in network protocols of the attacker's choice. Typically, the web attacker uses at least one machine as an HTTP server, which we refer to for simplicity as `attacker.com`. The web attacker can obtain SSL certificates for domains he or she owns; certificate authorities such as `instantssl.com` provide such certificates for free. The web attacker's network abilities are decidedly *weaker* than the usual network attacker considered in studies of network security because the web attacker can neither eavesdrop on messages sent to other recipients nor forge messages from other network locations. For example, a web attacker cannot act as a "man-in-the-middle."

- **Interaction with Client.** We assume the honest user views `attacker.com` in at least one browser window, thereby rendering the attacker's content. We make this assumption because we believe that an honest user's interaction with an honest site should be secure even if the user separately visits a malicious site in a different browser window. We assume the web attacker is constrained by the browser's security policy and does not employ a browser exploit to circumvent the policy. The web attacker's host privileges are decidedly *weaker* than an attacker who can execute a arbitrary code on the user's machine with the user's privileges. For example, a web attacker cannot install or run a system-wide key logger or botnet client.

Attacks accessible to a web attacker have significant practical impact because the attacks can be mounted without any complex or unusual control of the network. In addition, web attacks can be carried out by a standard man-in-the-middle network attacker, provided the user visits a single HTTP site, because a man-in-the-middle can intercept HTTP requests and inject malicious content into the reply, simulating a reply from `attacker.com`.

There are several techniques an attacker can use to drive traffic to `attacker.com`. For example, an attacker can place web advertisements, display popular content indexed by search engines, or send bulk e-mail to attract users. Typically, simply viewing an attacker's advertisement lets the attacker mount a web-based attack. In a previous study [20], we purchased over 50,000 impressions for $30. During each of these impressions, a user's browser rendered our content, giving us the access required to mount a web attack.

We believe that a normal, but careful, web user who reads news and conducts banking, investment, and retail transactions, cannot effectively monitor or restrict the provenience of all content rendered in his or her browser, especially in light of third-party advertisements. In other words, we believe that the web attacker threat model is an accurate representation of normal web behavior, appropriate for security analysis of browser security, and *not* an assumption that users promiscuously visit all possible bad sites in order to tempt fate.

**Gadget Attacker.** A *gadget attacker* is a web attacker with one additional ability: the integrator embeds a gadget of the attacker's choice. This assumption lets us accurately evaluate mashup isolation and communication protocols because the purpose of these protocols is to let an integrator embed untrusted gadgets safely. In practice, a gadget attacker can either wait for the user to visit the integrator or can redirect the user to the integrator's web site from `attacker.com`.

**Out-of-Scope Threats.** Although *phishing* [11, 7] can be described informally as a "web attack," the web attacker defined above does not attempt to fool the user by choosing a confusing domain name (such as `bankofthevvest.com`) or using other social engineering. In particular, we do *not* assume that a user treats `attacker.com` as if it were a site other than `attacker.com`. The attacks presented in this paper are "pixel-perfect" in the sense that the browser provides the user no indication whatsoever that an attack is underway. The attacks do not display deceptive images over the browser security indicators nor do they spoof the location bar and or the lock icon. In this paper, we do not consider *cross-site scripting* attacks, in which an attacker exploits a bug in an honest principal's web site to inject malicious content into another security origin. None of the attacks described in this paper rely on the attacker injecting content into another principal's security origin. Instead, we focus on privileges the browser itself affords the attacker to interact with honest sites.

## 3 Frame Isolation

Netscape Navigator 2.0 introduced the HTML `<frame>` element, which allows web authors to delegate a portion of their document's screen real estate to another document. These frames can be navigated independently of the rest of the main content frame and can, themselves, contain frames, further delegating screen real estate and creating a frame hierarchy. Most modern frames are embedded using the more-flexible `<iframe>` element, introduced in Internet Explorer 3.0. In this paper, we use the term *frame* to refer to both `<frame>` and `<iframe>` elements. The main, or *top-level*, frame of a browser window displays its location in the browser's location bar. Subframes are often indistinguishable from other parts of a page, and the browser does not display their location in its user interface. Browsers decorate a window with a lock icon only if every frame contained in the window was retrieved over HTTPS but do *not* require the frames to be served from the same host. For example, if `https://bank.com/` embeds a frame from `https://attacker.com/`, the browser will decorate the window with a lock icon.

**Organization.** Section 3.1 reviews browser security policies. Section 3.2 describes cross-window frame navigation attacks and defenses. Section 3.3 details same-window attacks that are not impeded by the cross-window defenses. Section 3.4 analyzes stricter navigation policies and advocates the "descendant policy." Section 3.5 documents our implementation and deployment of the descendant policy in major browsers.

## 3.1 Background

**Scripting Policy.** Most web security is focused on the browser's scripting policy, which answers the question "when is script in one frame permitted to manipulate the contents of another frame?" The scripting policy is the most important browser security policy because the ability to script another frame is the ability to control its appearance and behavior completely. For example, if `otherWindow` is another window's frame,

```
var stolenPassword =
    otherWindow.document.forms[0].
    password.value;
```

attempts to steal the user's password in the other window. Modern web browsers permit one frame to read and write all the DOM properties of another frame only when their content was retrieved from the same *origin*, i.e. when the scheme, host, and port number of their locations match. If the content of `otherWindow` was retrieved from a different origin, the browser's security policy will prevent this script from accessing `otherWindow.document`.

**Navigation Policy.** Every browser must answer the question "when is one frame permitted to navigate another frame?" Prior to 1999, all web browsers implemented a permissive policy:

| *Permissive Policy* |
|---|
| A frame can navigate any other frame. |

For example, if `otherWindow` includes a frame,

```
otherWindow.frames[0].location =
    "https://attacker.com/";
```

navigates the frame to `https://attacker.com/`. This has the effect of replacing the frame's document with content retrieved from that URL. Under the permissive policy, this navigation succeeds even if `otherWindow` contains content from a different security origin. There are a number of other idioms for navigating frames, including

```
window.open("https://attacker.com/",
            "frameName");
```

which requests that the browser search for a frame named `frameName` and navigate the frame to the specified URL. Frame names exist in a global name space and are not restricted to a single security origin.

**Top-level Frames.** Top-level frames are often exempt from the restrictions imposed by the browser's frame navigation policy. Top-level frames are less vulnerable to frame navigation attacks because the browser displays their location in the location bar. Internet Explorer and Safari do not restrict the navigation of top-level frames at all. Firefox restricts the navigation of top-level frames based on their *openers*, but this restriction can be circumvented [2]. Opera implements a number of restrictions on the navigation of top-level frames based on the current location of the frame.

## 3.2 Cross-Window Attacks

In 1999, Georgi Guninski discovered that the permissive frame navigation policy admits serious attacks [16]. Guninski discovered that, at the time, the password field on the CitiBank login page was contained within a frame. Because the permissive frame navigation policy lets any frame navigate any other frame, a web attacker can navigate the password frame on CitiBank's page to `https://attacker.com/`, replacing the frame with identical-looking content that sends the user's password to `attacker.com`. In the modern web, this *cross-window attack* might proceed as follows:

1. The user reads a popular blog that displays a Flash advertisement provided by `attacker.com`.

2. The user opens a new window to `bank.com`, which displays its password field in a frame.

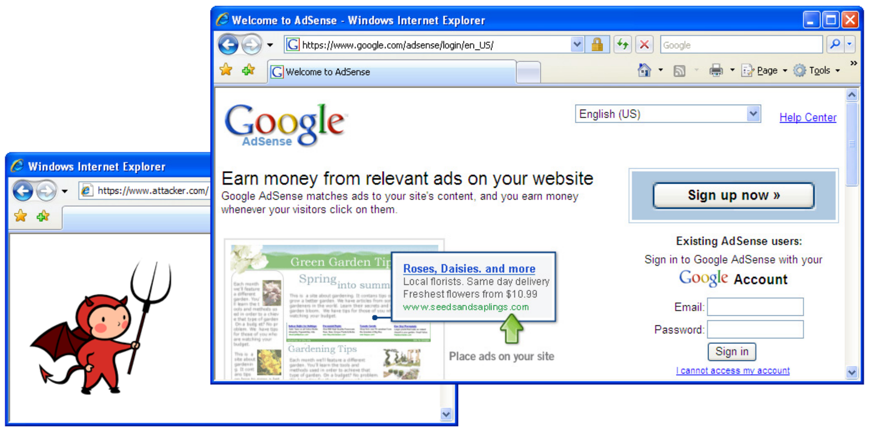
3. The malicious advertisement navigates the password frame to `https://attacker.com/`. The location bar still reads `bank.com` and the lock icon is *not* removed.

4. The user enters his or her password, which is then submitted to `attacker.com`.

Of the browsers in heavy use today, Internet Explorer 6 and Safari 3 both implement the permissive policy. Internet Explorer 7 and Firefox 2 implement stricter policies (described in subsequent sections). However, Flash Player can be used to circumvent the stricter navigation policy of Internet Explorer 7, effectively reducing the policy to "permissive." Many web sites are vulnerable to this attack, including Google AdSense, which displays its password field inside a frame; see Figure 1.

**Window Policy.** In response to Guninski's report, Mozilla implemented a stricter policy in 2001:

| *Window Policy* |
|---|
| A frame can navigate only frames in its window. |


Welcome to AdSense - Windows Internet Explorer
https://www.google.com/adsense/login/en_US/
Welcome to AdSense
Google AdSense
English (US)
Help Center
Sign up now »
Existing AdSense users:
Sign in to Google AdSense with your Google Account
Email:
Password:
Sign in
I cannot access my account
Place ads on your site
Windows Internet Explorer
https://www.attacker.com/

Earn money from relevant ads on your website
Google AdSense matches ads to your site's content, and you earn money whenever your visitors click on them.

Roses, Daisies, and more
Local florists. Same day delivery
Freshest flowers from $10.99
www.seedsandsaplings.com

Figure 1: Cross-Window Attack: The attacker controls the password field because it is contained within a frame.

This policy prevents the cross-window attack because the web attacker does not control a frame in the same window as the CitiBank or the Google AdSense login page. Without a foothold in the window, the attacker cannot navigate the login frame to `attacker.com`.

## 3.3 Same-Window Attacks

The window frame navigation policy is neither universally deployed nor sufficiently strict to protect users on the modern web because mashups violate its implicit security assumption that an honest principal will not embed a frame to a dishonest principal.

**Mashups.** A *mashup* combines content from multiple sources to create a single user experience. The party combining the content is called the *integrator* and the integrated content is called a *gadget*.

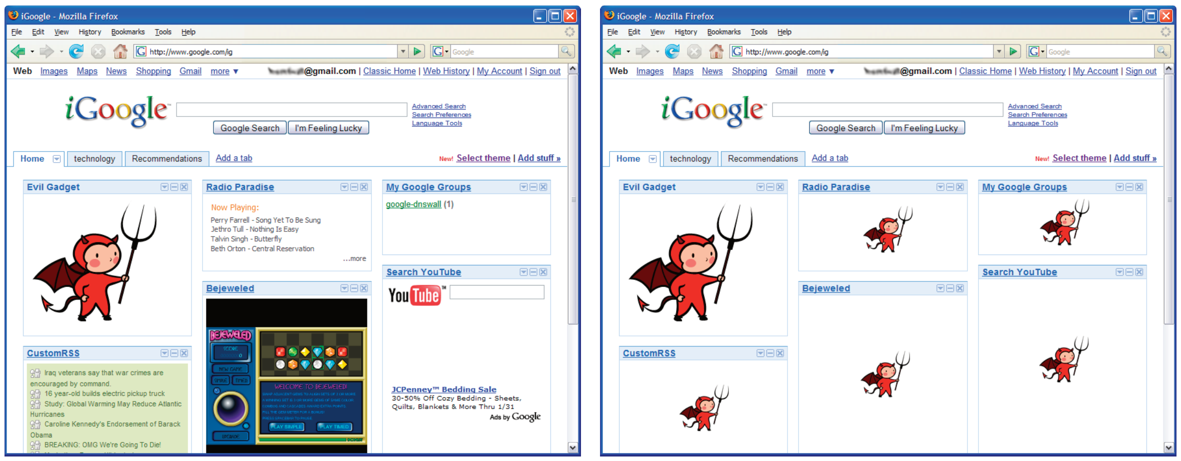
- **Aggregators.** Gadget aggregators, such as iGoogle [15], My Yahoo [40], and Windows Live [28], are one form of mashup. These sites let users customize their experience by selecting gadgets (such as stock tickers, weather predictions, news feeds, etc) to include on their home page. Third parties are encouraged to develop gadgets for the aggregator. These mashups embed the selected gadgets in a frame and rely on the browser's frame isolation to protect users from malicious gadgets.

- **Advertisements.** Web advertising is a simple form of mashup, combining first-party content, such as news articles or sports statistics, with third-party advertisements. Typically, the publisher (the integrator) delegates a portion of its screen real estate to an advertisement network, such as Google, Yahoo, or Microsoft, in exchange for money. Most advertisements, including Google AdWords, are contained in frames, both to prevent the advertisers (who provide the gadgets) from interfering with the publisher's site and to prevent prevent the publisher from using JavaScript to click on the advertisements.

We refer to aggregators and advertisements as *simple mashups* because these mashups do not involve communication between the gadgets and the integrator. Simple mashups rely on the browser to provide isolation but do not require inter-frame communication.

**Gadget Hijacking Attacks.** Mashups invalidate an implicit assumption of the window policy, that an honest principal will not embed a frame to a dishonest principal. A gadget attacker, however, does control a frame embedded by the honest integrator, giving the attacker the foothold required to mount a *gadget hijacking* attack [22]. In such an attack, a malicious gadget navigates a target gadget to `attacker.com` and impersonates the gadget to the user.

- **Aggregator Vulnerabilities.** iGoogle is vulnerable to gadget hijacking in browsers, such as Firefox 2, that implement the permissive or window policies; see Figure 2. Consider, for example, one popular iGoogle gadget that lets users access their Hotmail inbox. (This gadget is neither provided nor endorsed by Microsoft.) If the user is not logged into Hotmail, the gadget requests the user's Hotmail password. A malicious gadget can replace the Hotmail gadget with content that asks the user for his or her Hotmail password. As in the cross-window attack, the user is unable to distinguish the malicious password field from the honest password field.



(a) Before (b) After

Figure 2: Gadget Hijacking Attack. Under the window policy, the attacker gadget can navigate the other gadgets.

- **Advertisement Vulnerabilities.** Although text advertisements often do not contain active content (e.g., JavaScript), other forms of advertising, such as Flash advertisements, do contain active content. An attacker who provides such an advertisement can steal advertising impressions allotted to other advertisers via gadget hijacking. A malicious advertisement can traverse the page's frame hierarchy and navigate frames containing other advertisements to `attacker.com`, replacing the existing content with the attacker's advertisement.

## 3.4 Stricter Policies

Although browser vendors do not document their navigation policies, we were able to reverse engineered the navigation policies of existing browsers, and we confirmed our understanding with the browsers' developers. The existing policies are shown in Table 2. In addition to the permissive and window policies described above, we discovered two other frame navigation policies:

| *Descendant Policy* |
|---|
| A frame can navigate only its descendants. |

| *Child Policy* |
|---|
| A frame can navigate only its direct children. |

The Internet Explorer 6 team wanted to enable the child policy by default, but shipped the permissive policy because the child policy was incompatible with a large number of web sites. The Internet Explorer 7 team designed the descendant policy to balance the security requirement to defeat the cross-window attack with the compatibility requirement to support existing sites [33].

**Pixel Delegation.** The descendant policy provides the most attractive trade-off between security and compatibility because it is the least restrictive policy that respects pixel delegation. When one frame embeds another frame, the parent frame delegates a region of the screen to the child frame. The browser prevents the child frame from drawing outside of its bounding box but does allow the parent frame to draw over the child using the `position: absolute` style. The descendant policy permits a frame to navigate a target frame precisely when the frame could overwrite the screen real estate of the target frame. Although the child policy is stricter than the descendant policy, the additional strictness does not prevent many additional attacks because a frame can simulate the visual effects of navigating a grandchild frame by drawing over the region of the screen occupied by the grandchild frame. The child policy's added strictness does, however, reduce the policy's compatibility with existing sites, discouraging browser vendors from deploying the child policy.

**Origin Propagation.** A strict interpretation of the descendant policy prevents a frame from navigating its siblings, even if the frame is from the same security origin as its parent. In this situation, the frame can navigate its sibling indirectly by injecting script into its parent, which can then navigate the sibling because the sibling is a descendant of the parent frame. In general, browsers should decide whether or not to permit a navigation based on the active frame's security origin. Browsers should let an active frame navigate a target frame if there *exists* a frame in the same security origin as the active frame that has the target frame as a descendant. By recognizing this *origin propagation*, browsers can achieve a better trade-off between security and compatibly. These additional navigations do not sacrifice security because an attacker can perform the navigations indirectly, but allowing them is more convenient for honest web developers.

## 3.5 Deployment

We collaborated with the HTML 5 working group [18] and browser vendors to deploy the descendant policy in several browsers:

- **Safari.** We implemented the descendant policy as a patch for Safari. Apple accepted our patch and deployed the descendant policy to Mac OS X and Windows Safari users as a security update [30]. Apple also deployed our patch to all iPhone and iPod touch users.
- **Firefox.** We implemented the descendant policy as a patch for Firefox. Before accepting our patch, Mozilla requested tests for all their previous frame navigation regressions. We provided them with approximately 1000 lines of regression tests for their automatic test harness, covering the frame navigation security vulnerabilities from the past ten years. Mozilla accepted our patch and deployed the descendant policy in Firefox 3 [1].
- **Flash.** We reported to Adobe that Flash Player bypasses the descendant policy in Internet Explorer 7. Adobe agreed to ship a patch to all Internet Explorer users in their next security update.
- **Opera.** We notified Opera Software about inconsistencies in Opera's child policy that can be used in gadget hijacking attacks. They plan to fix these vulnerabilities in the upcoming release of Opera 9.5, and are evaluating the compatibility benefits of adopting the descendant policy [35].

## 4 Frame Communication

Over the past few years, web developers have built sophisticated mashups that, unlike simple aggregators and advertisements, are comprised of gadgets that communicate with each other and with their integrator. Yelp, which integrates the Google Maps gadget, motivates the need for secure inter-frame communication by illustrating how communicating gadgets are used in real deployments. Sections 4.1 and 4.2 analyze and improve fragment-identifier messaging and `postMessage`.

**Google Maps.** One popular gadget is the Google Maps API [14]. Google provides two mechanisms for integrating Google Maps:

- **Frame.** In the frame version of the gadget, the integrator embeds a frame to `maps.google.com`, which Google fills with a map centered at the specified location. The user can interact with map, but the integrator is oblivious to this interaction and cannot interact with the map directly.
- **Script.** In the script version of the gadget, the integrator embeds a `<script>` tag that executes JavaScript from `maps.google.com`. This script creates a rich JavaScript API the integrator can use to interact with the map, but the script runs with all of the integrator's privileges.

**Yelp.** Yelp is a popular review web site that uses the Google Maps gadget to display the locations of restaurants and other businesses it reviews. Yelp requires a high degree of interactivity with the Maps gadget because it places markers on the map for each restaurant and displays the restaurant's review when the user clicks on the marker. In order to deliver these advanced features, Yelp must use the script version of the Maps gadget. This design requires Yelp to trust Google Maps completely because Google's script runs with Yelp's privileges in the user's browser, granting Google the ability to manipulate Yelp's reviews and steal Yelp's customer's information. Although Google might be trustworthy, the script approach does not scale beyond highly respected gadget providers. Secure inter-frame communication provides the best of both alternatives: Yelp (and similar sites) can realize the interactivity of the script version of Google Maps gadget while maintaining the security of the frame version of the gadget.

## 4.1 The Fragment Identifier Channel

Although the browser's scripting policy isolates frames from different security origins, clever mashup designers have discovered an unintended channel between frames: the *fragment identifier channel* [3, 36]. This channel is regulated by the browser's less-restrictive frame navigation policy. This "found" technology lets mashup developers place each gadget in a separate frame and rely on the browser's security policy to prevent malicious gadgets from attacking the integrator and honest gadgets.

**Mechanism.** Normally, when a frame is navigated to a new URL, the browser retrieves the URL from the network and replaces the frame's document with the retrieved content. However, if the new URL different from the old URL only in the fragment (the portion after the `#`), then the browser does not reload the frame. If `frames[0]` is currently located at `http://example.com/doc`,

| IE 6 (default) | IE 6 (optional) | IE 7 (default) | IE 7 (optional) | Firefox 2 | Safari 3 | Opera 9 |
|---|---|---|---|---|---|---|
| Permissive | Child | Descendant | Permissive | Window | Permissive | Child |

Table 2: Frame navigation policies deployed in existing browsers.

```
frames[0].location =
    "http://example.com/doc#message";
```

changes the frame's location without reloading the frame or destroying its JavaScript context. The frame can observe the value of the fragment by periodically polling `window.location.hash` to see if the fragment identifier has changed. This technique can be used to send short string messages entirely within the browser, avoiding network latency. However, the communication channel is somewhat unreliable because, if two navigations occur between polls, the first message will be lost.

**Security Properties.** Because it was "found" and not designed, the fragment identifier channel has less-than-ideal security properties. The browser's scripting policy prevents security origins other than the one preceding the # from eavesdropping on messages because they are unable to *read* the frame's location (even though the navigation policy permits them to *write* to the frame's location). Browsers also prevent arbitrary security origins from tampering with portions of messages. Other security origins can, however, overwrite the fragment identifier in its entirety, leaving the recipient to guess the sender of each message.

To understand these security properties, we develop an analogy with well-known properties of network channels. We view the browser as guaranteeing that the fragment identifier channel has *confidentiality*: a message can be read only by its intended recipient. The fragment identifier channel fails to be a secure channel because it lacks *authentication*, the ability of the recipient to unambiguously determine the sender of a message. The channel also fails to be *reliable* because messages might not be delivered, and the attacker might be able to replay previous messages using the browser's `history` API.

The security properties of the fragment identifier channel are analogous to a channel on an untrusted network secured by a public-key cryptosystem in which each message is encrypted with the public key of its intended recipient. In both cases, if Alice sends a message to Bob, no one except Bob learns the contents of the message (unless Bob forwards the message). In both settings, the channel does *not* provide a reliable procedure for determining who sent a given message. There are two interesting differences between the fragment identifier channel and the public-key channel:

1. The public-key channel is susceptible to traffic analysis, but an attacker cannot determine the length of a message sent over the fragment identifier channel. An attacker can extract timing information by frequently polling the browser's clock, but obtaining a high-resolution timing signal significantly degrades the browser's performance.

2. The fragment identifier channel is constrained by the browser's frame navigation policy. In principle, this could be used to construct protocols secure for the fragment identifier channel that are insecure for the public-key channel (by preventing the attacker from navigating the recipient), but in practice this restriction has not prevented us from constructing attacks on existing protocol implementations.

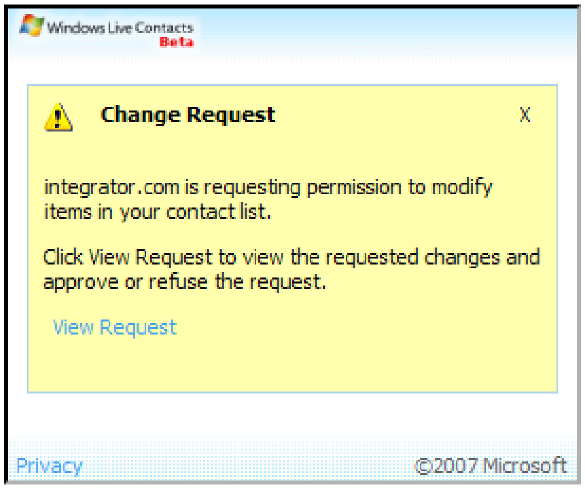
Despite these differences, we find the network analogy useful in analyzing inter-frame communication.

**Windows Live Channels.** Microsoft uses the fragment identifier channel in its Windows Live platform library to implement a higher-level channel API, `Microsoft.Live.Channels` [36]. The Windows Live Contacts gadget uses this API to communicate with its integrator. The integrator can instruct the gadget to add or remove contacts from the user's contacts list, and the gadget can send the integrator details about the user's contacts. Whenever the integrator asks the gadget to perform a sensitive action, the gadget asks the user to confirm the operation and displays the integrator's host name to aid the user in making trust decisions.

`Microsoft.Live.Channels` attempts to build a secure channel over the fragment identifier channel. By reverse engineering the implementation, we determined that it uses two sessions of the following protocol (one in each direction) to establish a secure channel:

$$\begin{aligned} A \to B &: N_A, \mathrm{URI}_A \\ B \to A &: N_A, N_B \\ A \to B &: N_B, \mathrm{Message}_1 \end{aligned}$$

In this notation, $A$ and $B$ are frames, $N_A$ and $N_B$ are fresh nonces (numbers chosen at random during each run of the protocol), and $\mathrm{URI}_A$ is the location of $A$'s frame. Under the network analogy described above, this protocol is analogous to a variant of the classic Needham-Schroeder key-establishment protocol [29].


Windows Live Contacts Beta

**Change Request**

integrator.com is requesting permission to modify items in your contact list.

Click View Request to view the requested changes and approve or refuse the request.

View Request

Privacy ©2007 Microsoft

Figure 3: Lowe Anomaly: This Windows Live Contacts gadget received a message that appeared to come from `integrator.com`, but in reality the request was made by `attacker.com`.

The Needham-Schroeder protocol was designed to establish a shared secret between two parties over an insecure channel. In the Needham-Schroeder protocol, each message is encrypted with the public key of its intended recipient. The Windows Live protocol does not employ encryption because the fragment identifier channel already provides the required confidentiality.

The Needham-Schroeder protocol has a well-known anomaly, due to Lowe [23], which leads to an attack in the browser setting. In the Lowe scenario, an honest principal, Alice, initiates the protocol with a dishonest party, Eve. Eve then convinces honest Bob that she is Alice. In order to exploit the Lowe anomaly, an honest principal must be willing to initiate the protocol with a dishonest principal. This requirement is met in mashups because the integrator initiates the protocol with the gadget attacker's gadget in order to establish a channel. The Lowe anomaly can be exploited to impersonate the integrator to the Windows Live Contacts gadget as follows:

$$\begin{aligned}
\text{Integrator} &\rightarrow \text{Attacker} : N_I, \text{URI}_I \\
\text{Attacker} &\rightarrow \text{Gadget} : N_I, \text{URI}_I \\
\text{Gadget} &\rightarrow \text{Integrator} : N_I, N_G \\
\text{Integrator} &\rightarrow \text{Attacker} : N_G, \text{Message}_1
\end{aligned}$$

After these four messages, the attacker possesses $N_I$ and $N_G$ and can impersonate the integrator to the gadget. We have successfully implemented this attack against the Windows Live Contacts gadget. The issue is easily observable for the Contacts gadget because the gadget displays the integrator's host name to the user in its security user interface; see Figure 3.

**SMash and OpenAjax 1.1.** A recent paper [22] from IBM proposed another protocol for establishing a secure channel over the fragment identifier channel. They describe their protocol as follows:

> The SMash library in the mashup application creates the secret, an unguessable random value. When creating the component, it includes the secret in the fragment of the component URL. When the component creates the tunnel iframe it passes the secret in the same manner.

The SMash developers have contributed their code to the OpenAjax project, which plans to include their fragment identifier protocol in version 1.1. The SMash protocol can be understood as follows:

$$\begin{aligned}
A &\rightarrow B : N, \text{URI}_A \\
B &\rightarrow A : N \\
A &\rightarrow B : N, \text{Message}_1
\end{aligned}$$

This protocol admits the following simple attack:

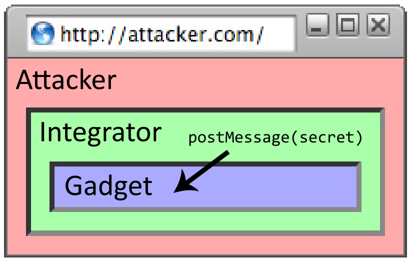
$$\begin{aligned}
\text{Attacker} &\rightarrow \text{Gadget} : N, \text{URI}_I \\
\text{Gadget} &\rightarrow \text{Integrator} : N \\
\text{Attacker} &\rightarrow \text{Gadget} : N, \text{Message}
\end{aligned}$$

We have confirmed this attack by implementing the attack against the SMash implementation. Additionally, the attacker is able to conduct this attack covertly by blocking the message from the gadget to the integrator because the message waits for the `load` event to fire.

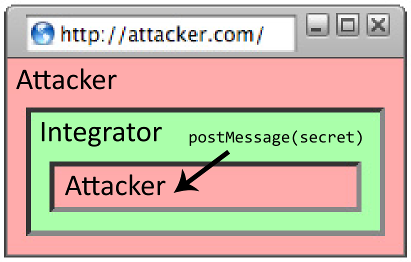
**Secure Fragment Messaging.** The fragment identifier channel can be secured using a variant of the Needham-Schroeder-Lowe protocol [23]. The main idea in Lowe's improvement of the Needham-Schroeder protocol is that the responder must include his identity in the second message of the protocol, letting the honest initiator determine that an attack is in progress and abort the protocol.

$$\begin{aligned}
A &\rightarrow B : N_A, \text{URI}_A \\
B &\rightarrow A : N_A, N_B, \text{URI}_B \\
A &\rightarrow B : N_B \\
&\dots \\
A &\rightarrow B : N_A, N_B, \text{Message}_i \\
B &\rightarrow A : N_A, N_B, \text{Message}_j
\end{aligned}$$

We contacted Microsoft, IBM, and the OpenAJAX Alliance about the vulnerabilities in their fragment identifier messaging protocols and suggested the above protocol improvement. Microsoft adopted our suggestions and deployed a patched version of



(a) Integrator sends secret messages to child



(b) Attacker hijacks integrator's child

Figure 4: Recursive Mashup Attack

`Microsoft.Live.Channels` and of the Windows Live Contacts gadget. IBM adopted our suggestions and revised their SMash paper. The OpenAJAX Alliance adopted our suggestions and updated their codebase. All three now use the above protocol to establish a secure channel using fragment identifiers.

## 4.2 The postMessage Channel

HTML 5 [19] specifies a new browser API for asynchronous communication between frames. Unlike the fragment identifier channel, the `postMessage` channel was designed for cross-site communication. The `postMessage` API was originally implemented in Opera 8 and is now supported by Internet Explorer 8, Firefox 3 [37], and Safari [24].

**Mechanism.** To send a message to another frame, the sender calls the `postMessage` method:

```
frames[0].postMessage("Hello world.");
```

The browser then generates a `message` event in the recipient's frame that contains the message, the origin (scheme, port, and domain) of the sender, and a JavaScript pointer to the frame that sent the message.

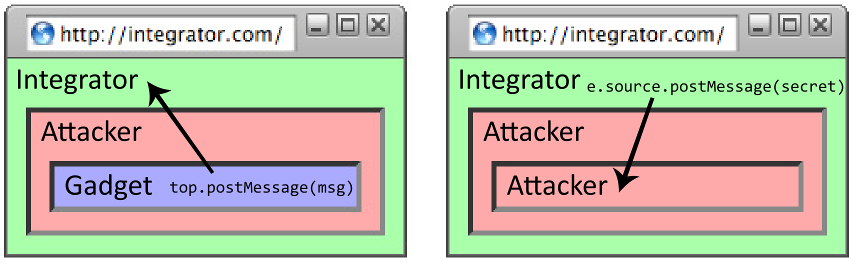
**Security Properties.** The `postMessage` channel guarantees *authentication*, messages accurately identify their senders, but the channel lacks confidentiality. Thus, `postMessage` has almost the "opposite" security properties as the fragment identifier channel. Where the fragment identifier channel has confidentiality without authentication, the `postMessage` channel has authentication without confidentiality. The security properties of the `postMessage` channel are analogous to a channel on a untrusted network secured by an existentially unforgeable signature scheme. In both cases, if Alice sends a message to Bob, Bob can determine unambiguously that Alice sent the message. With `postMessage`, the `origin` property accurately identifies the sender; with cryptographic signatures, verifying the signature on a message accurately identifies the signer of the message. One difference between the channels is that cryptographic signatures can be easily replayed, but the `postMessage` channel is resistant to replay attacks. In some cases, however, an attacker might be able to mount a replay attack by reloading honest frames.

**Attacks.** Although `postMessage` is widely believed to provide a secure channel between frames, we show an attack on the confidentiality of the channel. A message sent with `postMessage` is directed at a frame, but if the attacker navigates that frame to `attacker.com` before the `message` event is generated, the attacker will receive the message instead of the intended recipient.

- **Recursive Mashup Attack.** Suppose, for example, that an integrator embeds a frame to a gadget and then calls `postMessage` on that frame. The attacker can load the integrator inside a frame and carry out an attack without violating the descendant frame navigation policy. After the attacker loads the integrator inside a frame, the attacker navigates the gadget frame to `attacker.com`. Then, when the integrator calls `postMessage` on the "gadget's" frame, the browser delivers the message to the attacker whose content now occupies the "gadget's" frame; see Figure 4. The integrator can prevent this attack by "frame busting," i.e., by refusing to render the mashup if `top !== self`, indicating that the integrator is contained in a frame.

- **Reply Attack.** Another `postMessage` idiom is also vulnerable to interception, even under the child frame navigation policy:

```
window.onmessage = function(e) {
  if (e.origin == "https://b.com")
    e.source.postMessage(secret);
};
```



(a) Gadget requests secret from integrator (b) Integrator's reply is delivered to attacker

Figure 5: Reply Attack

> The `source` attribute of the `MessageEvent` is a JavaScript reference to the frame that sent the message. It is tempting to conclude that the reply will be sent to `https://b.com`. However, an attacker might be able to intercept the message. Suppose that the honest gadget calls `top.postMessage("Hello")`. The gadget attacker can intercept the message by embedding the honest gadget in a frame, as depicted in Figure 5. After the gadget posts its message to the integrator, the attacker navigates the honest gadget to `https://attacker.com`. (This navigation is permitted under both the child and descendant frame navigation policies.) When the integrator replies to the `source` of the message, the message will be delivered to the attacker instead of to the honest gadget.

**Securing postMessage.** It might be feasible for sites to build a secure channel using `postMessage` as an underlying communication primitive, but we would prefer that `postMessage` provide a secure channel natively. In MashupOS [39], we proposed a new browser API, `CommRequest`, to send messages between origins. When sending a message using `CommRequest`, the sender addresses the message to a principal:

```
var req = new CommRequest();
req.open("INVOKE",
         "local:https://b.com//inc");
req.send("Hello");
```

Using this interface, `CommRequest` protects the confidentiality of messages because the `CommServer` will deliver messages only to the specified principal. Although `CommRequest` provides adequate security, the `postMessage` API is further along in the standardization and deployment process. We therefore propose extending the `postMessage` API to provide the additional security benefits of `CommRequest` by including a second parameter: the origin of the intended recipient. If the sender specifies a target origin, the browser will deliver the message to the targeted frame *only if* that frame's current security origin matches the argument. The browser is free to deliver the message to any principal if the sender specifies a target origin of `*`. Using this improved API, a frame can reply to a message using the following idiom:

```
window.onmessage = function(e) {
  if (e.origin == "https://b.com")
    e.source.postMessage(secret,
                         e.origin);
};
```

As shown in this example use, the API uses the same origin syntax for both sending and receiving messages. The scheme is included in the origin for those developers who wish to defend against active network attackers by distinguishing between HTTP and HTTPS. We implemented this API change as a patch for Safari and a patch for Firefox. Our proposal was accepted by the HTML 5 working group [17]. The new API is now included in Firefox 3 [38], Safari [32], and Internet Explorer 8 [25].

## 5 Related Work

**Mitigations for Gadget Hijacking.** SMash [22] mitigates gadget hijacking (which the authors refer to as "frame phishing") without modifying the browser by carefully monitoring the frame hierarchy and browser events for evidence of unexpected navigation. Neither the integrator nor the gadget can prevent these navigations, but the mashup can alert the user and refuse to function if it detects an illicit navigation. This approach lets an attacker mount a denial-of-service attack against the mashup, but a web attacker can already mount a denial-of-service attack against the entire browser by issuing a blocking `XMLHttpRequest` or entering an infinite loop.

Unfortunately, this approach can lead to false positives. SMash waits 20 seconds for a gadget to load before assuming that the gadget has been hijacked and warning the user. An attacker might be able to fool the user into entering sensitive information during this time interval. Using a shorter time interval might cause users with slow network connections to receive warnings even though no attack is in progress. We expect that the deployment of the descendant policy will obviate the need for server-enforced gadget hijacking mitigations.

**Safe Subsets of HTML and JavaScript.** One way to sidestep the security issues of frame-based mashups is to avoid using frames entirely and render the gadgets together with the integrator in a single document. This approach forgoes the protections of the browser's security policy because all the gadgets and the integrator share a single browser security context. To maintain security, this approach requires gadgets to be written in a "safe subset" of HTML and JavaScript that prevents a malicious gadget from attacking the integrator or other gadgets. Analyzing the security and usability of these subsets is an active area of research. Several open-source [13, 4] and closed-source [31, 10] implementations are available. FBML [10] is currently the most successful of these subsets and is used by millions of users as the foundation of the Facebook Platform.

Writing programs in one of these safe subsets is often awkward because the language is highly constrained to avoid potentially dangerous features. To improve usability, the safe subsets are often accompanied by a compiler that transforms untrusted HTML and JavaScript into the subset, possibly at the cost of performance. These safe subsets will become easier to use over time as these compilers become more sophisticated and more libraries become available, but with the deployment of `postMessage` and the descendant policy, we expect that frame-based mashup designs will continue to find wide use as well.

**Other Frame Isolation Proposals.** There are several other proposals for frame isolation and communication:

- **Subspace.** In Subspace [21], we used a multi-level hierarchy of frames that coordinated their `document.domain` property to communicate directly in JavaScript. Similar to most frame-based mashups, the descendant frame navigation policy is required to prevent gadget hijacking.
- **Module Tag.** The proposed `<module>` tag [5] is similar to an `<iframe>` tag, but the module runs in an unprivileged security context, without a principal, and the browser prevents the integrator from overlaying content on top of the module. Unlike `postMessage`, the communication primitive used with the module tag is intentionally unauthenticated: it does not identify the sender of a message. It is unknown whether navigation can be used to intercept messages as there are no implementations of the `<module>` tag.
- **Security=Restricted and Jail.** Internet Explorer supports a `security` attribute [26] of frames that can be set to `restricted`. With `security="restricted"`, the frame's content cannot run JavaScript. Similarly, the proposed `<jail>` tag [8] encloses untrusted content and prevents the sandboxed content from running JavaScript. However, eliminating JavaScript prevents gadgets from offering interactive experiences.
- **MashupOS.** Our MashupOS proposal [39] includes new primitives for isolating web content while allowing secure communication. Our improvements to `postMessage` and frame navigation policies allow web authors to obtain some of the benefits of MashupOS using existing web APIs.

## 6 Conclusions

Web browsers provide a platform for web applications. These applications rely on the browser to isolate frames from different security origins and to provide secure inter-frame communication. To provide isolation, browsers implement a number of security policies, including a frame navigation policy. The original frame navigation policy, the permissive policy, admits a number of attacks. The modern frame navigation policy, the descendant policy, prevents these attacks by permitting one frame to navigate another only if the frame could draw over the other frame's region of the screen. The descendant policy provides an attractive trade-off between security and compatibility, is deployed in the major browsers, and has been standardized in HTML 5.

In existing browsers, frame navigation can be used as an inter-frame communication channel with a technique known as fragment identifier messaging. If used directly, the fragment identifier channel lacks authentication. To provide authentication, `Windows.Live.Channels`, SMash, and OpenAjax 1.1 use messaging protocols. These protocols are vulnerable to attacks on authentication but can be repaired in a manner analogous to Lowe's variation of the Needham-Schroeder protocol [23].

The `postMessage` communication channel suffered the converse security vulnerability: using frame navigation, an attacker can breach the confidentiality of the channel. We propose providing confidentiality by extending the `postMessage` API to let the sender specify an intended recipient. Our proposal was adopted by the HTML 5 working group, Internet Explorer 8, Firefox 3, and Safari.

With these improvements to the browser's isolation and communication primitives, frames are a more attractive feature for integrating third-party web content. Two challenges remain for mashups incorporating untrusted content. First, a gadget is permitted to navigate the top-level frame and can redirect the user from the mashup to a site of the attacker's choice. This navigation is made evident by the browser's location bar, but many users ignore the location bar. Improving the usability of the browser's security user interface is an important area of future work. Second, a gadget can subvert the browser's security mechanisms if the attacker employs a browser exploit to execute arbitrary code. A browser design that provides further isolation against this threat is another important area of future work.

## Acknowledgments

We thank Mike Beltzner, Sumeer Bhola, Dan Boneh, Gabriel E. Corvera, Ian Hickson, Koji Kato, Eric Lawrence, Erick Lee, David Ross, Maciej Stachowiak, Hallvord Steen, Peleus Uhley, Jeff Walden, Sam Weinig, and Boris Zbarsky for their helpful suggestions and feedback. This work is supported by grants from the National Science Foundation and the US Department of Homeland Security.

## References

[1] Adam Barth et al. Adopt "descendant" frame navigation policy to prevent frame hijacking. `https://bugzilla.mozilla.org/show_bug.cgi?id=408052`.

[2] Adam Barth and Collin Jackson. Protecting browsers from frame hijacking attacks, December 2007. `http://crypto.stanford.edu/frames/`.

[3] James Burke. Cross domain frame communication with fragment identifiers. `http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html`.

[4] Douglas Crockford. ADsafe: Making JavaScript safe for advertising. `http://adsafe.org/`.

[5] Douglas Crockford. The `<module>` tag. `http://www.json.org/module.html`.

[6] Neil Daswani, Micheal Stoppelman, et al. The anatomy of Clickbot.A. In *Proc. HotBots*, 2007.

[7] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on human factors in computing systems*, 2006.

[8] Brendan Eich. JavaScript: Mobility and ubiquity. `http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf`.

[9] Facebook. Badges. `http://www.facebook.com/help.php?page=4`.

[10] Facebook. Facebook Markup Language (FBML). `http://wiki.developers.facebook.com/index.php/FBML`.

[11] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An Internet con game. In *Proceedings of the 20th National Information Systems Security Conference*, 1996.

[12] Flickr API. `http://flickr.com/services/api/`.

[13] Google. Caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`.

[14] Google. Google Maps API. `http://code.google.com/apis/maps/`.

[15] iGoogle. `http://www.google.com/ig`.

[16] Georgi Guninski. Frame spoofing using loading two frames. `https://bugzilla.mozilla.org/show_bug.cgi?id=13871`.

[17] Ian Hickson. Re: A potential slight security enhancement to postMessage, Februrary 2008. `http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-February/013949.html`.

[18] Ian Hickson. Re: HTML5 frame navigation policy, April 2008. `http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-April/014597.html`.

[19] Ian Hickson et al. HTML 5 Working Draft. `http://www.whatwg.org/specs/web-apps/current-work/`.

[20] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[21] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference. (WWW)*, 2007.

[22] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure cross-domain mashups on unmodified browsers. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, 2008.

[23] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055. Springer Verlag, 1996.

[24] Henry Mason. No support for MessageEvent interface, 2007. `https://bugs.webkit.org/show_bug.cgi?id=14994`.

[25] Microsoft. postMessage method. `http://msdn.microsoft.com/en-us/library/cc197015(VS.85).aspx`.

[26] Microsoft. SECURITY attribute. `http://msdn2.microsoft.com/en-us/library/ms534622(VS.85).aspx`.

[27] Microsoft. Try the Windows Live Contacts control. `http://dev.live.com/mashups/trypresencecontrol/`.

[28] Microsoft. Windows Live. `http://home.live.com/`.

[29] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[30] National Institute of Standards and Technology. CVE-2007-5858, December 2007.

[31] Charlie Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[32] Adam Roben et al. Change postMessage/MessageEvent to match HTML5 wrt. exposing origin vs. domain/uri. `https://bugs.webkit.org/show_bug.cgi?id=17331`.

[33] David Ross, 2008. Personal communication.

[34] J. Ruderman. JavaScript Security: Same Origin. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[35] Hallvord Steen, 2008. Personal communication.

[36] Danny Thorpe. Secure cross-domain communication in the browser. *The Architecture Journal*, 12:14–18, July 2007. `http://msdn2.microsoft.com/en-us/library/bb735305.aspx`.

[37] Jeff Walden. Implement HTML5's cross-document messaging API (postMessage), 2007–2008. `https://bugzilla.mozilla.org/show_bug.cgi?id=387706`.

[38] Jeff Walden et al. Update postMessage and MessageEvent to reflect domain/uri being replaced by origin, optional origin argument. `https://bugzilla.mozilla.org/show_bug.cgi?id=417075`.

[39] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[40] Yahoo! My Yahoo! `http://my.yahoo.com/`.

# Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking

Michael Martin
*Computer Science Department*
*Stanford University*
*mcmartin@cs.stanford.edu*

Monica S. Lam
*Computer Science Department*
*Stanford University*
*lam@cs.stanford.edu*

## Abstract

Cross-site scripting (XSS) and SQL injection errors are two prominent examples of taint-based vulnerabilities that have been responsible for a large number of security breaches in recent years. This paper presents QED, a goal-directed model-checking system that automatically generates attacks exploiting taint-based vulnerabilities in large Java web applications. This is the first time where model checking has been used successfully on real-life Java programs to create attack sequences that consist of multiple HTTP requests.

QED accepts any Java web application that is written to the standard servlet specification. The analyst specifies the vulnerability of interest in a specification that looks like a Java code fragment, along with a range of values for form parameters. QED then generates a goal-directed analysis from the specification to perform session-aware tests, optimizes to eliminate inputs that are not of interest, and feeds the remainder to a model checker. The checker will systematically explore the remaining state space and report example attacks if the vulnerability specification is matched.

QED provides better results than traditional analyses because it does not generate any false positive warnings. It proves the existence of errors by providing an example attack and a program trace showing how the code is compromised. Past experience suggests this is important because it makes it easy for the application maintainer to recognize the errors and to make the necessary fixes. In addition, for a class of applications, QED can guarantee that it has found all the potential bugs in the program. We have run QED over 3 Java web applications totaling 130,000 lines of code. We found 10 SQL injections and 13 cross-site scripting errors.

## 1 Introduction

As more and more business applications migrate to the Web, the nature of the most dangerous threats facing users has changed. Web applications are typically written in languages that make classic exploits like buffer overruns impossible, but new infrastructures bring new vulnerabilities. Two of the most popular attacks in this domain are SQL injection and cross site scripting (XSS) [12]. This paper presents a practical, programmable technique that can automatically generate attacks for large web-based applications. The system also shows the statements executed over the course of the attack. This information can be used by application developers to close these security holes.

Many commercial systems, including Cenzic's Hailstorm [7] and Core Security's Core Impact [9], rely on *black-box testing*. In black-box testing of web applications, the tester only has the level of access available to any external attacker—that is, it may only make HTTP requests and examine the responses. This approach has the advantage that any such analysis is independent of the target application's implementation language, making it ideal for broad deployment. However, it cannot take advantage of the logic of the program; it may not be efficient, and it cannot provide any guarantee on coverage.

This paper presents a system called QED that automatically finds attack vectors for a large class of vulnerabilities in web applications written in the same application framework. This system is based on the approach of *concrete model checking*. This is a verification technique based on systematic exploration of a program's state space. It is an attractive approach to security problems because not only can it conclusively find vulnerabilities, if a systematic exploration proves exhaustive, it can prove that no vulnerabilities exist. However, this technique is generally not feasible for large, real-life programs. In addition, a web application continuously accepts inputs, so it seems impossible on the surface to exhaust all possible paths. To make QED a practical tool that works on real programs, we built the system based on the design principles listed below.

1. Many web application vulnerabilities, such as SQL injection and cross-site scripting, can be generalized as taint-based problems. By focusing on this class rather than one vulnerability at a time, the QED system is much more general. Users can specify taint-based vulnerabilities in a language called PQL [22]. In fact, PQL extends beyond even taint-based analysis as it includes execution patterns involving any sequence of methods on a set of objects that is describable via a context-free language.

   Users can use QED for finding different vulnerabilities, and even vulnerabilities that are specific to their own applications. It is very important that ordinary developers be able to generate these analyses on their own.

2. Today, application frameworks are heavily used in web application development as they greatly reduce software engineering time. We advocate extending the notion of frameworks beyond software development to include code auditing. Exploiting higher level semantic information about the framework makes it possible to generate more effective static analyses. Furthermore, by abstracting away the guts of a framework, we can concentrate our model checker's effort on the application code itself. This abstraction step needs only to be performed once for each framework, as the abstracted code is reusable. For this research, we have picked the following popular core frameworks for web applications:

   - Java servlets [27], which is a standard extension to the Java platform for writing web applications.
   - JSPs (Java Server Pages) [28], which allow page design to be commingled with database accesses.
   - Apache Struts [1], which is a web application framework that uses the model-view-controller paradigm. In this paradigm, a controller decouples the data model from the user view so they can easily be changed independently.

   *Any* Java web application intended for deployment in a standard application server conforms to the servlet specification. If a Java web application also uses JSP or Struts, our framework will take advantage of the additional semantics as well.

   To demonstrate the effectiveness of this approach, we report the result of applying our tool across three different Java web applications developed on this framework.

3. In model checking, we are simulating the program execution on candidate input sequences. QED uses JPF, the Java PathFinder model checking system [29], to do this. It is important that we concentrate the model checking time on sequences that are likely to identify vulnerabilities. Based on the query, QED automatically compiles user-supplied queries into static analyses for the web application that prune out input sequences that are guaranteed not to expose any vulnerability. The static analysis generates a set of input vectors. If it is small, this set can be tested exhaustively; if it is not, the static analysis's results—directed by the user's query—direct the checker to test more promising results first.

## 1.1 Contributions

This paper makes the following contributions.

- *A session-based model for user input in web applications.* Much work in testing web applications focuses on either analyzing individual pages [31] or simulating a browser user with a sophisticated spider [3]. We present a technique that bases its user model on data flow information across requests in a session. This helps restrict the search space while also exposing possible vulnerabilities that a spider or nonmalicious end user might never produce.

- *A programmable approach to checking event-driven applications.* QED is extremely flexible; its concept of vulnerability is merely "anything that matches a specification", and the permissible specifications include any context-free language of method calls on a consistent set of run-time objects. Though this paper focuses on taint vulnerabilities in web applications, the technique generalizes to other error patterns as well as other event-based systems such as GUI applications or file systems.

- *A model-checking framework to systematically explore standard Java web applications.* We have implemented a simulated environment for the Java PathFinder model checker that will systematically explore programs based on the Java Servlet Specification. We have refined it further to work more effectively with the popular Apache Struts framework.

- *Experimental validation of our approach.* We supplied specifications for two major security vulnerabilities (cross-site scripting and SQL injections) and applied the QED system to three large Web applications. These applications totaled roughly 130,000 lines of non-library code. QED detected 10 SQL injection vulnerabilities and 13 XSS vulnerabilities.

## 1.2 Paper Organization

Section 2 describes the class of vulnerabilities of interest. Section 3 describes how we apply model checking to web applications to generate the attack vectors and get the execution trace. Section 4 describes how we use static analysis to reduce the search space of model checking. Section 5 demonstrates the QED algorithm step by step on an example application. Section 6 details experimental results. Section 7 discusses related work, and Section 8 concludes.

## 2 Problem Statement

Our algorithm accepts a web application and a vulnerability specification, then generates a set of attack path components with corresponding execution traces. This section describes the class of applications and vulnerabilities our system addresses.

## 2.1 Taint Vulnerabilities

SQL injection and cross-site scripting are both instances of *taint vulnerabilities*. All such vulnerabilities are detected in a similar manner: untrusted data from the user is tracked as it flows through the system, and if it flows unsafely into a security-critical operation, a vulnerability is flagged. In SQL injection, the user can add additional conditions or commands to a database query, thus allowing the user to bypass authentication or alter data. With XSS, an attacker can inject his own HTML (including JavaScript or other executable code) into a web page; this is exploitable in many ways, up to complete compromise of the browser. In the so-called "reflection attack" [12] XSS is used by a phisher to inject credential-stealing code into official sites without having to redirect the user to a copy of the site. This means that any security credentials will be valid on the attack site, and even whitelisting will not prevent the attack.

Given the gravity of the vulnerabilities, we would like to eliminate their existence before deploying our applications. Some of these vulnerabilities can be subtle, however. It is not sufficient to just consider URLs in isolation because an attack may consist of a sequence of URLs. Consider a scenario with the example web application in Figures 1 and 2. An attack on this application can go as follows: the attacker sends the victim an email containing the URL `http://example.com/search_begin.jsp?s=<script...` where the `s` parameter carries a JavaScript payload crafted to log users' keyboard entries. The victim clicks on the link. Since this is the user's first interaction with `example.com`, a new session is created by the web server, and when the JSP checks the value of `login`, it finds nothing. It thus stores

```
<html>
<head>
<% HttpSession s = getSession();
   if (s.getAttribute("login") == null) {
     s.setAttribute("text",
                    getParameter("s"); %>
  <meta http-equiv="refresh"
      content="10;URL=search_login.jsp">
</head>
<body></body>
</html>
<% } else { %>
<!-- rest of page... -->
```

Figure 1: Snippet from `search_begin.jsp`.

```
<html>
<body>
<h1>Login required</h1>
<p>To search for
   <%=getSession().getAttribute("text")%>,
   you must first log in.</p>
<form>
<!-- rest of page... -->
```

Figure 2: Snippet from `search_login.jsp`.

the search string in the session and generates a redirect page to `search_login.jsp`. That page then generates an error and requests login information. However, at this point it echoes the value from the session blindly, thus injecting the script and allowing the attacker to log the user's password. This example illustrates that we need to analyze more than just individual requests to be sure we have found all vulnerabilities in a web application.

We model the behavior of a web application as a series of request-response events; each URL corresponds to an HTTP request, and this request is processed to produce a response. We may characterize an attack vector by a sequence of URL requests in a session where untrusted input data propagates into security-critical operations.

## 2.2 Domain of Web Applications

We model a web application as a reactive system that operates on a *session* at a time. A session consists of a series of *events*, with each event being an HTTP request submitted by the same user. Note that while the request originates from the same user, its contents may actually be manipulated by an attacker. We do not place any restriction on the ordering of events. In particular, it is not necessary that requests be constrained by the links available on the last page viewed. This is necessary because an attacker can construct and send malicious requests directly. This also argues against using web-spider techniques to collect potential attack vectors.

In response to an event, a web application may modify the *session data*. This is information that is user-specific but maintained temporarily on the webserver over the course of a user's interaction with the machine. In a webserver, a separate data structure is normally maintained for each user, and cookies or special arguments would be set to match each users to their sessions.

Sessions are assumed to be independent of each other. An attack may consist of a sequence of events within a session, but cannot span multiple sessions. Our reasoning here is that any attack usable against another user should also be usable against oneself, and so the attack will still manifest.

## 2.3 Vulnerability Specifications

The set of taint-based vulnerabilities addressed by our technique consists of all attacks that match the following pattern:

1. Untrusted data is read in from some *taint source*, such as a user-controlled file, URL request, cookie value, or network source. It may subsequently be stored in arbitrary objects and passed in and out as parameters or returned results.

2. Some methods may derive new objects from old. Some of these, if passed an untrusted object, will produce an untrusted object. Examples include methods that parse a request and create subobjects from the untrusted data, or methods that create larger strings by appending characters to the untrusted data. We call these methods *propagators*.

3. No untrusted data, whether from the original taint source or derived via propagators, may be used in any *taint sink*, such as a database access routine.

4. The previous rule does not apply if the object has been passed through one of several *sanitizers*, that quote or escape the contents of the object.

This is an abstraction of the general problem of *information flow control*. Information is tracked from the source, through propagators, until it either hits a sanitizer and becomes safe, or hits a taint sink and possibly does damage. Once the tracker can confirm that all dangerous data only reaches sanitizers, a proof of the correctness of these sanitizers will suffice to prove the correctness of the entire program.

Our vulnerability specification consists of four patterns, one for each of the previously enumerated components. These patterns are expressed as PQL queries. PQL is a powerful specification language that permits one to specify patterns of events on objects in a manner similar to program snippets. It permits subqueries to be defined and then matched against as well. We can exploit this by defining the components of our specification as subqueries and then linking them together with a generic main query that works for any taint problem.

```
query source(object * x)
matches
    HttpServletRequest.getParameter(x)
 |  x = Cookie.getValue();

query prop(object * x, object *y)
matches
    (StringBuilder) y.append(x)
 |  y = (StringBuilder) x.toString();


query sink(object * x)
matches
    JspWriter.print*(x)
 |  JspWriter.write(x, ...);
```

Figure 3: XSS vulnerability specification.

A simple example for XSS in JSPs is shown in Figure 3. All three of its defined subqueries are a logical OR between individual method calls. Its taint sources, `HttpRequest.getParameter` and `Cookie.getValue`, are defined for all Java web applications [27]. Likewise, the `JspWriter` class in the taint sink is defined in the JSP specification [28]. PQL permits method names to be regular expressions, and so we collect all `print` and `println` method calls within a single clause.

The propagation rules in the `prop` query handle string concatenation in Java 1.5. In the full specification, other versions of Java and other modes of string propagation are also handled. These are simply added as additional OR clauses; we omit them here for clarity.

Care must be taken when developing the specification—missing a propagator may lead to false negatives in the final result, while missing sanitizers is likely to lead to many false positives. A suitably crafted general specification, however, can apply to many applications directly or with only minor modifications to specify details and application-specific sanitizers. Furthermore, the operation of the model checker will suggest which modifications need to be made to refine the query.

Due to the design of the Java libraries, web application queries will rarely need to explicitly specify sanitizers. Java's `String` class is immutable, and it is also the class that represents the beginning and end points of any web transaction. Since the sanitization process will generally create an entirely new `String`, this freshly created object would thus be considered safe. This is another reason we must be particularly careful not to miss any propagators: any propagator we fail to specify will be treated as a san-

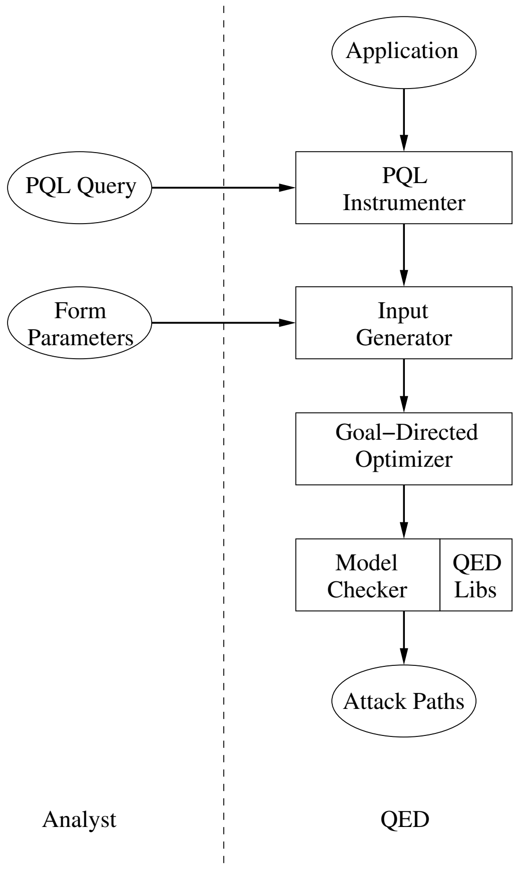


Figure 4: QED architecture. User-supplied information is on the left.

itizer.

It is also possible that a sanitizer might perform its transforms using propagator methods. This would require explicitly marking the result as sanitized. However, this situation never occurred in our experiments. We never found it necessary to explicitly specify sanitizers, and our XSS query worked unmodified with all applications.

## 2.4 PQL Instrumentation and Matching

The vulnerability specification is translated by the PQL compiler into a set of instrumentation directives. When applied to the target application, they weave in monitoring code to detect matches to the query, and to report on the objects involved [22]. When a match is found by the monitor, it signals the model checker to report that a failure condition has been found. If no match occurs, the model checker's backtracking mechanism will also roll back the matching machinery to the appropriate state.

## 3 Input Generation

In this section, we describe how QED enumerates attack vectors for a target web application. An analyst must provide two components: the PQL query specifying the vulnerability, and a set of input values for any form parameters. Given these, QED will do the rest. A diagram of the process is shown in Figure 4.

The input application is first instrumented according to the provided PQL query, as described in Section 2.4. The instrumented application is then combined with a custom, automatically generated harness. This is a program that will systematically explore the space of URL requests. Each URL consists of a page request (the *path*, covered in Section 3.1), and an optional set of input parameters (the *query*, discussed in Section 3.2). The harnessed application is then fed to the model checker, along with stub implementations of the application server's environment. The results of that model checker correspond directly to sequences of URLs that demonstrate the attack paths.

We may also optionally improve our search by optimizing the harness before the model checking step; we discuss these refinements in Section 4.

## 3.1 Generating Page Requests

An attack path is a sequence of URLs, each of which consists of a page request (the *path*) and a set of input parameters (the *query)* [4]. The web application translates a URL into a method invocation with a set of parameters.

Thus, a URL corresponding to our sample JSP earlier:

$$\texttt{http://www.example.com/search.jsp?s = foo}$$

would translate into the method invocation

$$\texttt{org.apache.jsp.search\_jsp.doGet(req, resp)}$$

where the call `req.getParameter(x)` yields the value "`foo`" if `x` is "`s`", and yields `null` otherwise. The `resp` parameter represents the response to be returned.

There is a simple correspondence between a URL and a method invocation. We refer to the method invocation as an *event*. An *event* consists of:

1. a reference to an event handler. The event handler corresponds to the path of the URL. It identifies the name of the Java method to be invoked when a matching URL is received.

2. event handler parameters. These typically correspond to the query part of the URL. They provide extra parameters used by the handler, and generally carry the more free-form data. They may include

cookies or information supplied by a user when filling out forms. They may also contain the payload that an attacker wishes to inject into the system. Thus, it is very important to model these inputs carefully.

Most Java web applications are developed using a framework that makes explicit the set of URLs it accepts and its corresponding event handlers. Our system currently handles three popular frameworks, as discussed below.

- *Servlets* are the most basic form of server-side Java, and are the lowest level of abstraction available. Any Java web application intended to be run in a standard application server must ultimately use this specification. Individual servlets are Java classes that implement a well-specified API [27]. The URL-to-servlet mapping is specified by an XML file as part of the application's metadata. QED simply interprets the XML file to determine the list of event handlers in the application.

- *Java Server Pages*, or *JSPs*, provide a PHP-like interface to Java [28]. They are compiled by a JSP compiler such as Jasper into servlets. The URL-to-servlet mapping in this case is specified by a transformation of the JSP's path in the file system, which generates the class name.

- *Apache Struts* is a popular application platform built on top of JSPs and the core servlet specification [1, 13]. It implements its own `Action` API similar to the servlets API, but which forwards to JSP files for actual HTML output. A URL in a Struts application thus maps to two calls in sequence; a call to an `Action`'s entry point, and a call to the associated JSP's entry point. These mappings from URLs to `Actions` and JSPs are specified in an XML file in a manner similar to the specifications for servlets.

For each of these, QED can produce a comprehensive list of paths understood by the application. To test each sequence, it does, by default, a breadth-first search through them - first checking all sequences of length 1, then all of length 2, and so forth. This has no obvious termination condition, however; our optimizations and heuristics in Section 4 provide limits.

## 3.2 Parameters to Event Handlers

In Java web applications, data from the user is represented by a set of key-value pairs mapping strings to strings. Applications conforming to the Java Servlet Specification use a method called `getParameter` to retrieve a value for a given key. QED rewrites methods corresponding to taint sources to call out to the model-checker, indicating to the model checker that there is non-determinism associated with the returned value of the method. The model checker will cycle through the possible values, including the option that no such key was provided by the user.

We rely on the analyst to provide a sufficient pool of values to test the application. It would be infeasible to test every possible string that could be supplied to the event handlers, but it is also not necessary. Our goal is merely to show that it is possible for data from a taint source to reach a taint sink. If a controlled string is displayed, this is a vulnerability.

In cases where the contents of an input string do matter, the data are often expected to be in a certain form: if they do not conform to the expected type, some paths may not be executable. For our experiments, we supplied one of the common default types used by web applications in general: integers, booleans ("yes", "true", etc.) and generic strings. We also included the null object to represent the lack of an argument.

Applications may also require application-specific "magic" values that influence control flow. The most common case for this is an `action` variable or similar, which holds one of several values depending on the value of a list box or similar. In such cases, QED can usually extract the information we need via a constant propagation analysis; this will tell us if an argument from the query string is compared against constant strings. By enumerating these strings and ensuring they are possible values for our keys, we search the input space more exhaustively.

It would be possible to combine this work with an analysis similar to EXE [6] to determine a set of inputs that would exercise all predicates in the web application. For our experiments so far, however, we have found that even our simple constant-propagation analysis is overkill. Almost all data read from the user is processed and dumped directly into a data sink. In these circumstances the control flow cannot change based on input.

## 4 Goal-Directed Optimization

In this section, we present several optimizations to reduce the search space of model checking. The key insight is that the we should not treat all URL sequences as equally likely to yield a new vulnerability, since we may have already checked a shorter, equivalent sequence. Since we check in increasing order of length, any match it finds will have already been discovered. There are four principles we apply to focus the search:

- *The final request in the sequence must finish the demonstration of a vulnerability* (Section 4.1).

- *Every request must, directly or indirectly, influence the final result* (Section 4.2).
- *No sequence ever repeats a request* (Section 4.3).
- *A match can only occur in a sequence if there are objects that would satisfy that match participating in that sequence* (Section 4.4).

## 4.1 Filtering Final Events

QED's model checker searches through candidate sequences in length order. This means that for any given vulnerability in the code, the shortest demonstration of it will appear first. If it does not, any possible vulnerability would have already been shown before the final request was processed, so a prefix of the sequence would suffice, and will in fact have already been checked. This condition is thus stronger than a simple breadth-first search, which can only confidently eliminate sequences with a prefix corresponding to a known vulnerability.

To perform the final event filter, we need two pieces of information. First, we need to know which method calls in the application can in fact complete a match. For a taint problem, this is straightforward, as it is any method listed as a taint sink. For PQL in general it may be necessary to perform a simple control-flow analysis on the query to determine the set of events that can occur last.

We then need to determine which URL requests can lead to match completion. We do this by writing a simple harness program that calls each entry point in the application in turn. We then compute a call graph of this harness and determine which entry points can eventually call a match-completing method.

Any sequence which does not end in a call to one of these entry points is guaranteed to not affect the final result, and thus may be discarded.

## 4.2 Eliminating Redundant URL Sequences

HTTP is a stateless protocol. Web applications maintain state across requests either client-side with cookies or server-side with session data. We treat cookies as a source of user input, as cookie information may be forged, deleted mid-session, or otherwise tampered with. Session information remains under the control of the server and can thus be tracked more precisely.

The motivation behind this optimization is that this mechanism is the sole form of data-flow through the session. If there is no data-flow contributed by a part of a candidate sequence, we need not include that part. Furthermore, since we are checking in increasing order of length, removing this redundant part of the sequence produces a sequence that we have already either checked or proven irrelevant.

To perform this optimization we need a way to characterize the cross-request data-flow. We do this via a *dependence relation*: an event handler $m_1$ *depends* on another event handler $m_2$ if $m_1$ can potentially read the data written by $m_2$. To compute the dependence relation, we must determine the flow of data within a session.

The Java Servlet Specification provides an explicit API to capture this. Data are passed between handlers via a special object of type `javax.servlet.HttpSession`. This session object functions as a string-to-object map. For each request, we determine what string values can be used as keys to the map for reads and writes. This information is available via a call graph analysis as in Section 4.1, supplemented with pointer and constant-propagation information to determine which string values may be used as keys. If a nonconstant string is used as a key, we assume that handler may access anything in the session.

With this information we can compute the dependence relation by treating each key as a storage location and determining def-use information. We then take the transitive closure of the dependence relation, and eliminate any sequence in which there are requests that do not influence the final request.

## 4.3 Removing Repetitive Cycles

If the dependency relation is cyclic, there will be a countably infinite number of possible candidates to test. To keep the test sequence finite, we restrict our sequences to only call any given entry point once.

This heuristic would need to be refined for web applications where one physical page serves as multiple logical pages (controlled, say, by some `action` parameter); however, this situation did not arise in any of our experiments.

## 4.4 Statically Eliminating Sequences

We further reduce the search space by using a static analysis to prune off sequences that cannot possibly match our query. This is especially important for sequences that use a large number of widely variable parameters, as eliminating a single sequence can translate into thousands or even millions of candidates that need not be checked. The algorithm is described below.

1. QED constructs a new harness for the application that iterates through all sequences that pass the preceding three criteria. The harness defines a method for each input sequence, and the method calls the

entry point for each of the URL request in the sequence.

2. QED translates the PQL query specifying the defect of interest into a sound context-sensitive interprocedural analysis that determines if the query can be satisfied. QED applies the analysis to the harness to find the methods (input sequences) that can potentially generate a match. The algorithm used has been been described in a previous paper [22]. This analysis tracks pointers in a context-sensitive but flow-insensitive manner. The analysis is sound—no approximation done by the pointer analysis will produce false negatives. All sequences found by the analysis to be incapable of generating a match may be ignored without compromising the soundness of the model checker.

The success of this step hinges on both the precision and the conservativeness of the pointer analysis used. An overly imprecise analysis will not be able to eliminate any candidates, while a non-conservative analysis will prune away candidates that might be valid. The QED system applies the context-sensitive, conservative, interprocedural, and inclusion-based analysis of Whaley and Lam [32], along with improvements by Livshits et al. to handle reflection [21]. The results of this analysis are stored in a deductive database which QED consults throughout the optimization process [19].

## 5 Example

We will now show the operation of this algorithm by detecting an XSS vulnerability in a simple three-page web application. The pages in this application are as follows:

- `search.jsp`, which presents a search form to the user and sends the results on to `searching.jsp`.
- `searching.jsp`, which reads a search parameter `s` and stores it in the session. The display is a simple timed redirect to `result.jsp`.
- `result.jsp`, which prints the results of the search. It also echoes the initial input, retrieved from the session. This represents a cross-site scripting vulnerability.

For our example, we use the stock XSS vulnerability query from Figure 3. The PQL instrumenter will transform the application, tracking all calls to sources, sinks, and propagators.

For our model environment, we will only concern ourselves with whether or not an argument is present, so we will set `null` and "`SampleString`" as our input pool. QED will generate a test harness for the application, providing these values as plausible results for the sources, and calling all possible sequences of events. Since we only concider non-repeating sequences, there are ten: three of length 1, six of length 2, and one of length 3. The entry points for these events will simply be the `doGet` methods on the classes corresponding to each JSP.

In the optimization step, the final events filter has no effect for this query. The sink for the XSS query is `JspWriter.print()`, which all three pages call as part of their output generation.

The dependency criterion is much more fruitful. Our session-based def-use analysis concludes that `searching.jsp` writes the session, while `result.jsp` reads it with the same key. This yields a dependency relation with one fact, and the dependency criterion eliminates all but four sequences—each page alone, and the [`searching.jsp`, `result.jsp`] sequence. Factoring in the choice of `s` in `searching.jsp`, this yields a grand total of five test runs.

The pointer analysis phase shows that `searching.jsp` is the only request handler with a source in it, thus eliminating two of the length-one sequences immediately. It can then show that, as `searching.jsp`'s parameter is only fed into a session and the handler itself only emits constant strings, the lone `searching.jsp` request also cannot complete a match. Thus, for our example application, we are able to pre-prune every sequence of events but one. The only task remaining for the model checker is to demonstrate which values for `s`, if any, will actually produce a vulnerability.

The model checker will return the following sequence as a demonstration of an XSS attack path:

- `searching.jsp?s = SampleString`
- `result.jsp`

Despite the fact that a typical use case would derive its input from `search.jsp`, the page does not actually contribute anything to the vulnerability itself.

In general, the amount of search space that can be removed by our optimizations will depend on several factors. The number and prevalence of taint sinks is one; if there are more places where the path can end, there will clearly be more paths. However, the dominant factor will be the fan-out from the session data-flow. With a low fan-out, even a large number of sinks will not multiply unduly.

## 6 Experimental Results

We applied QED to three Struts-based web applications from the open-source repository Sourceforge. Basic information about these is shown in Figure 5. They are

| Benchmark | Description | Lines of Code | Classes | Event Handlers | Dependency Pairs |
|---|---|---|---|---|---|
| PersonalBlog | Blogging software | 17,149 | 132 | 15 | 0 |
| JOrganizer | Address book | 31,897 | 263 | 46 | 49 |
| JGossip | Forum system | 79,685 | 556 | 80 | 267 |

Figure 5: Applications used in the experiments. (The lines of code do not include library classes)

| Benchmark | Non-redundant URL Sequences | Ends in SQL Sink | SQL Sessions | SQL Errors | XSS Sessions | XSS Errors |
|---|---|---|---|---|---|---|
| PersonalBlog | 15 | 2 | 2 | 2 | 1 | 1 |
| JOrganizer | 356,358 | 260 | 153 | 8 | 86 | 3 |
| JGossip | 1,062,539 | 16,031 | 9,436 | 0 | 30 | 9 |

Figure 6: Analysis results.

listed in order of their size. For each application, we list the number of classes defined in the program, the size of the application itself (not counting library classes), and the total number of event handlers specified by the application's deployment metadata. The last column of Figure 5 shows the number of dependency pairs found by our dependence analysis described in Section 4.2.

We used QED to locate both cross-site scripting and SQL injection vulnerabilities in each of these applications. Each of these applications depends on a database backend. The JGossip application used JDBC directly; the other two used object persistence libraries that we modeled as stubs. All three applications, since they are Struts-based, rely on JSPs for their output, and so the XSS analysis dealt primarily with those.

Figure 6 presents some measurements of our experiment. The first column (Non-redundant URL Sequences) lists the number of sessions whose URLs are not repeated and not redundant according to their data dependencies. Personalblog does not have cycles in its dependence graph, so it is possible to exhaustively model check the program by testing the specified number of input sequences. The next column (Ends in SQL Sink) shows the result of applying the full redundancy elimination analysis algorithm presented in Section 4.2. The next column (SQL Sessions) shows the number of sessions that needs to be checked after the feasibility analysis from Section 4.4 is also taken into account. The next column gives the number of SQL injections QED discovered.

The final two columns provide similar information for XSS. We do not provide an equivalent to the "Ends in SQL Sink" column because the XSS sink is HTML output, and so every HTTP response by definition includes a sink. Between SQL injection and cross-site scripting, we thus cover both rare and common sinks in our applications.

For comparison, even if we restrict ourselves to non-repeating URL sequences, the naïve approach of Section 3 would test a number of sessions proportional to the factorial of the number of event handlers. In JGossip, this is approximately $10^{120}$ sequences.

## 6.1 PersonalBlog

The PersonalBlog system is a web application based on Struts and the Hibernate 2 object persistence system [2]. It makes no interesting use of session objects, so there are no dependencies between handlers. Thus, the dependence analysis shows that we can consider each event handler in isolation without compromising any guarantee on security. Since there are only 15 event handlers in the program, and each request has few parameters, the model checker can run through all the cases quickly.

QED found one XSS attack vector and two SQL attack vectors. Note that a single vector can have multiple vulnerabilities. In this case, one of the SQL vectors has two SQL injection possibilities. Thus, there are actually three SQL vulnerabilities that we have found. The static analysis in this case was accurate in identifying all the vulnerabilities, without generating any false positives. The model checker generates the input vectors and a program execution trace showing the details of their existence.

The results of running PQL itself, as a dynamic checker, on PersonalBlog has also been reported previously [22]. Not only did QED find all the vulnerabilities previously identified, it found an additional one. This discrepancy is due to QED having a more inclusive specification than in the previous work, tracking information from HTTP headers and not just from the URL proper.

## 6.2 JOrganizer

JOrganizer is a personal contact and appointment manager of moderate size. Access to the backing database is managed within the application by an "Object Query Language" that reduces directly to SQL, much like Hibernate 2.

The application has 46 event handlers in total. The dependence analysis shows that there are 49 pairs of dependent event handlers. The dependence relations are cyclic, which means that we will have to restrict our attention to acyclic sequences to keep the test space finite.

QED then further focuses the model checking effort by using information specific to each vulnerability. We found that 15 of the event handlers cannot touch the database at all, and thus cannot be final events for SQL injection. Furthermore, none of the single-event sequences exhibits a SQL vulnerability. The reason is that no event is allowed to touch the database unless it is preceded by a "log-in" event. Our analysis shows a dependence between these events and the "log-in" event. QED ignores the independent pairs, and keeps testing sequences with first a log-in event and then a database access event.

QED is able to iterate through all the filtered, non-redundant, and non-repeating sequences in this case, finding three XSS vulnerabilities and eight SQL vulnerabilities.

## 6.3 JGossip

JGossip is a large application with nearly 80,000 lines of code in 80 actions. There are many cyclic dependencies among event handlers in JGossip. Even if we restrict the sequences under consideration to non-redundant and non-repeating events, over a million sequences still remain. Furthermore, within these sequences, many requests used enormous numbers of input parameters. One event had 15 parameters, which, with a pool of 5 possible inputs per parameter, would generate over 30 billion test cases simply for that one URL. For event handlers such as those we restricted our model checker's input pool to two possibilities per parameter, lowering the number of test cases per handler to a more manageable 32,000.

Next QED tries to reduce the number of candidate vectors based on the vulnerability specification. For SQL injection, the taint sink method is database queries. A majority of the 80 actions touch the database. However, our static feasibility analysis shows that only seven of these database accesses may touch tainted objects. Thus we have only 7 final events to consider. Of the seven, five have no dependency chains longer than length two. They are responsible for a total of 37 potential attack vectors, and they are all of length 2. The remaining two have many dependencies, and the static analyzer can only narrow them to 9,436 candidate attack vectors. Once parameters are factored in, this still yields hundreds of millions of candidates to check, so there are still too many to consider. At the end, we managed to check only all the seven sequences with a single taint-sink event, and the all 37 sequences of length 2. The model checker found no SQL vulnerability.

As it happens, this lack of SQL injections is unsurprising, because JGossip is constructed to be independent of its database backend. As such, all of its database requests are ultimately constant strings; it uses a hash table to look up which strings are appropriate for the appropriate action, based on the SQL dialect used by the backend. Since all SQL queries end up being constant strings, this suggests that no injections are possible; however, its use of hash tables forced the program analyzer to make conservative approximations on seven of the actions, thus leading to the need for a model checking step.

The tests for XSS were much more straightforward; all of the actions corresponding to possibly dangerous output JSPs had few inputs and few dependencies, leading to a grand total of only 30 sessions to check. The XSS vulnerabilities so found were also located immediately, since session data did not affect their outputs.

## 6.4 Experimental Summary

The three web applications in our experimental study illustrate a spectrum of effects we can get with QED. PersonalBlog shows an example where QED is able to prove that there are no vulnerabilities other than the ones found. By proving that the events have no dependencies, QED can simply check the URLs one at a time. JOrganizer shows that in the presence of dependencies, our analyses can greatly improve the effectiveness of model checking and provides good coverage. QED was able to check all the sequences without repeated URLs. Lastly, JGossip shows that model checking for really large programs remains a challenge. The static analyzer is useful as a way of directing the model checking to focus on sequences with higher payoffs.

## 7 Related Work

Systematic automated testing is not entirely novel, but it is also not commonplace. Our work was informed by both the FiSC system [34] and WebSSARI [17]. WebSSARI's approach is much different from QED's, in that it focuses on abstract interpretation of PHP code looking for violations of data flow control. QED, on the other hand, owes more of its design philosophy to FiSC. FiSC operated in an entirely different problem domain (filesystem correctness) and simply searched for evidence of errors rather than the cause. Its implementation was based on the CMC model checker [23] which is also much closer to our JPF-based system than WebSSARI's runtime solution.

The techniques described in this paper touch upon a wide variety of disciplines. Model checking is the most directly obvious of these. Our system uses the Java PathFinder system [29]. JPF was suitable for our system primarily due to the ability to directly run sizable Java applications as bytecode; this permitted us to treat our dynamic analysis as just another part of the application being checked. Classical model checkers such as SPIN [14] require a special specification language which abstracts the application greatly. Other model checking systems such as Bandera [8] also directly abstract the Java source, which complicates its utility for our purposes.

The more general field of bugfinding comprises an enormous amount of work. In recent years, web applications have received a good deal of attention due to their unique vulnerabilities and flaws. SABER is a static tool that detects flaws based on pattern templates [25]. Livshits and Lam made progress in creating a sound analysis on web applications that produced a usably low false positive rate [20]. The WebSSARI system, in its pre-model-checking work, allows the specification of taint-style data-flow problems on PHP-based applications, and systematically searches for dangerous information flows [16]. Nguyen-Tuong et al. use similar approaches, also for PHP [24]. In a more general context FindBugs attempts to locate a broad class of bugs in Java applications of all kinds [15], and the Metal system let the user specify state machines to represent error conditions [11]. The SQLCHECK system uses a much more precise technique to detect grammatical changes in commands as a result of user input [26]. The QED system provides a general analysis that the user specializes, while SQLCHECK is SQL-injection specific and FindBugs is a battery of unrelated analyses. Taint flow within an application is tracked incidentally, and only if the PQL specification demands it.

Our characterization of inputs, when combined with model checking, can be seen as a form of testing, and all testing techniques perform better with a better set of inputs. Some work has been done on systematically deducing inputs that will explore the state space of an application. Systems such as Korat [5] attempt to systematically produce only consistent inputs; this is rarely relevant to web applications, whose arguments can be nearly-arbitrary strings. Korat's general principle of deducing input sets from execution constraints, however, may still be applicable. Symbolic execution techniques, such as DART [10] and EXE [33], suitably adapted to deal with string and URL data, are more likely to be a fruitful adjunct to the techniques in this paper. Some work has been done already to provide these techniques for JPF but the results given seem to indicate that at present it scales only to smaller applications [30].

For the specific problem of cross-site scripting, recent work has focused on extending the DOM to permit browser extensions to block out any unauthorized scripts [18]. While, if fully implemented, this system will block out any possible attacks, it requires cooperation between both site authors and clients. Client-side protection is also of limited use against taint problems such as SQL injection that attack the server.

## 8 Summary and Conclusions

Security concerns regarding web applications are here to stay, and likely only to grow in importance. Cross-site scripting and SQL injection are two of the most popular kinds of vulnerabilities. This paper presented a technique called goal-directed model checking that can find attack vectors for these vulnerabilities automatically and efficiently. Armed with actual attack vectors and their corresponding execution trace, it is easier to convince the developers that it is necessary to change the code, and also to pinpoint how the problem can be fixed.

Our technique is implemented in a system called QED. Users can use the system for any taint-based vulnerability on Java applications developed using servlets, JSPs, or Struts. We applied QED to three programs and found errors in every one of them, yielding a total of 10 SQL injection and 13 XSS vulnerabilities. This result is worrisome, suggesting that there are plenty of security risks in using web applications.

This work also shows for the first time how we can combine techniques from three approaches to generate a useful and powerful system:

*Sound, sophisticated program analysis*. Sophisticated analysis based on context-sensitive pointer alias analysis is precise enough to use on production software, despite being conservative to retain soundness. Nonetheless, false positives are still bound to occur with a conservative analysis.

*Dynamic monitoring*. Dynamic analysis does not have false positives, but it can only spot problems that its input happens to trigger.

*Model checking*. Model checking has many advantages: it executes all the paths in a program; it has no false positives; it has no false negatives with respect to the set of possible inputs tried; it identifies actual attack vectors; and it can generates an execution trace for any input. However, it is too slow.

QED combines the advantages of all the three approaches. It uses sound analysis to optimize both dynamic monitoring and model checking, dynamic monitoring to follow the flow of taint, and finally model checking to generate the actual attack vectors.

Cross-site scripting and SQL injection are examples of errors that exist at the application layer and that are not due to simple language deficiencies like buffer overruns. We can expect to see many more varieties of errors that operate at this higher semantic level. This suggests that programmable systems like bddbddb, PQL, and QED are important so that developers can utilize the technology, without being analysis experts, for their own programs.

The widespread adoption of application frameworks in software development opens up a new opportunity for managing software complexity. These software frameworks should come with testing, model checking, static analysis, and dynamic monitoring submodules; they should be programmable and specialized for that framework. Perfecting them as part of the framework will put these advanced technologies in the hands of many more developers.

## 9 Acknowledgements

We are grateful to Benjamin Livshits and Christopher Unkel for their reviews of previous versions of this draft, and to Willem Visser, Peter Mehlitz, the rest of the JPF4 team, the TRUST project, and the anonymous reviewers for their encouragement and suggestions. This work was supported in part by the National Science Foundation under Grant No. UCB-0424422 and Grant No. 0326227.

## References

[1] Apache Software Foundation. Apache Struts. `http://struts.apache.org`, 2002.

[2] Bauer, C., and King, G. *Hibernate in Action.* Manning Publications, New York, NY, 2004.

[3] Benedikt, M., Friere, J., and Godefroid, P. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of the Alternate Track of the 11th International World Wide Web Conference (WWW'02)* (2002).

[4] Berners-Lee, T., Fielding, R., and Masinter, L. RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. `http://www.ietf.org/rfc/rfc2396.txt`, 1998.

[5] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated Testing Based on Java Predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002), pp. 123–133.

[6] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006).

[7] Cenzic. Hailstorm. `http://www.cenzic.com/`.

[8] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. Bandera: Extracting Finite-State Models from Java Source Code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering* (2000), pp. 439–448.

[9] Core Security. Core impact overview. `http://www.coresecurity.com/?module=ContentMod&action=item&id=32`.

[10] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2005), pp. 213–223.

[11] Hallem, S., Chelf, B., Xie, Y., and Engler, D. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 69–82.

[12] Hoglund, G., and McGraw, G. *Exploiting Software: How to Break Code.* Addison-Wesley Publishing, 2004.

[13] Holmes, J. *Struts: The Complete Reference.* McGraw-Hill/Osborne, Emeryville, CA, 2004.

[14] Holzmann, G. J. The Model Checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[15] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. In *Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2004), pp. 132–136.

[16] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th Conference on the World Wide Web* (2004), pp. 40–52.

[17] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Verifying Web Applications Using Bounded Model Checking. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN2004)* (2004), pp. 199–208.

[18] Jim, T., Swamy, N., and Hicks, M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)* (2007), pp. 601–610.

[19] Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. Context-Sensitive Program Analysis as Database Queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2005), ACM Press, pp. 1–12.

[20] Livshits, V. B., and Lam, M. S. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium* (Aug. 2005), pp. 271–286.

[21] Livshits, V. B., Whaley, J., and Lam, M. S. Reflection Analysis for Java. In *APLAS'05 - the Third Asian Symposium on Programming Languages and Systems* (2005), pp. 139–160.

[22] Martin, M. C., Livshits, B., and Lam, M. S. Finding Application Errors and Security Flaws using PQL: a Program Query Language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005), pp. 365–383.

[23] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Conference on Operating System Design and Implementation (OSDI)* (2002), pp. 75–88.

[24] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC)* (2005), pp. 295–308.

[25] Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R. D., Kershenbaum, A., and Koved, L. SABER: Smart Analysis Based Error Reduction. In *Proceedings of International Symposium on Software Testing and Analysis* (2004), pp. 243–251.

[26] SU, Z., AND WASSERMANN, G. The Essence of Command Injection Attacks in Web Applications. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), pp. 372–382.

[27] SUN MICROSYSTEMS. JSR-000154 Java Servlet 2.5 Specification. `http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html`, 2004.

[28] SUN MICROSYSTEMS. JSR-000245 JavaServer Pages 2.1. `http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html`, 2006.

[29] VISSER, W., HAVELUND, K., BRAT, G., PARK, S.-J., AND LERDA, F. Model Checking Programs. *Automated Software Engineering 10*, 2 (2003), 203–232.

[30] VISSER, W., PĂSĂREANU, C. S., AND KHURSID, S. Test Input Generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 97–107.

[31] WASSERMAN, G., AND SU, Z. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)* (2007), pp. 32–41.

[32] WHALEY, J., AND LAM, M. S. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)* (2004).

[33] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2006), pp. 234–248.

[34] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using Model Checking to Find Serious File System Errors. In *Proceedings of the Conference on Operating System Design and Implementation (OSDI)* (2004), pp. 273–288.

# Lest We Remember: Cold Boot Attacks on Encryption Keys

J. Alex Halderman*, Seth D. Schoen†, Nadia Heninger*, William Clarkson*, William Paul‡, Joseph A. Calandrino*, Ariel J. Feldman*, Jacob Appelbaum, and Edward W. Felten*

*Princeton University †Electronic Frontier Foundation ‡Wind River Systems

{jhalderm, nadiah, wclarkso, jcalandr, ajfeldma, felten}@cs.princeton.edu
schoen@eff.org, wpaul@windriver.com, jacob@appelbaum.net

## Abstract

Contrary to popular assumption, DRAMs used in most modern computers retain their contents for several seconds after power is lost, even at room temperature and even if removed from a motherboard. Although DRAMs become less reliable when they are not refreshed, they are not immediately erased, and their contents persist sufficiently for malicious (or forensic) acquisition of usable full-system memory images. We show that this phenomenon limits the ability of an operating system to protect cryptographic key material from an attacker with physical access. We use cold reboots to mount successful attacks on popular disk encryption systems using no special devices or materials. We experimentally characterize the extent and predictability of memory remanence and report that remanence times can be increased dramatically with simple cooling techniques. We offer new algorithms for finding cryptographic keys in memory images and for correcting errors caused by bit decay. Though we discuss several strategies for partially mitigating these risks, we know of no simple remedy that would eliminate them.

## 1 Introduction

Most security experts assume that a computer's memory is erased almost immediately when it loses power, or that whatever data remains is difficult to retrieve without specialized equipment. We show that these assumptions are incorrect. Ordinary DRAMs typically lose their contents gradually over a period of seconds, even at standard operating temperatures and even if the chips are removed from the motherboard, and data will persist for minutes or even hours if the chips are kept at low temperatures. Residual data can be recovered using simple, nondestructive techniques that require only momentary physical access to the machine.

We present a suite of attacks that exploit DRAM remanence effects to recover cryptographic keys held in memory. They pose a particular threat to laptop users who rely on disk encryption products, since an adversary who steals a laptop while an encrypted disk is mounted could employ our attacks to access the contents, even if the computer is screen-locked or suspended. We demonstrate this risk by defeating several popular disk encryption systems, including BitLocker, TrueCrypt, and FileVault, and we expect many similar products are also vulnerable.

While our principal focus is disk encryption, any sensitive data present in memory when an attacker gains physical access to the system could be subject to attack. Many other security systems are probably vulnerable. For example, we found that Mac OS X leaves the user's login password in memory, where we were able to recover it, and we have constructed attacks for extracting RSA private keys from Apache web servers.

As we discuss in Section 2, certain segments of the computer security and semiconductor physics communities have been conscious of DRAM remanence effects for some time, though strikingly little about them has been published. As a result, many who design, deploy, or rely on secure systems are unaware of these phenomena or the ease with which they can be exploited. To our knowledge, ours is the first comprehensive study of their security consequences.

**Highlights and roadmap** In Section 3, we describe experiments that we conducted to characterize DRAM remanence in a variety of memory technologies. Contrary to the expectation that DRAM loses its state quickly if it is not regularly refreshed, we found that most DRAM modules retained much of their state without refresh, and even without power, for periods lasting thousands of refresh intervals. At normal operating temperatures, we generally saw a low rate of bit corruption for several seconds, followed by a period of rapid decay. Newer memory technologies, which use higher circuit densities, tended to decay more quickly than older ones. In most cases, we observed that almost all bits decayed at predictable times and to predictable "ground states" rather than to random values.

We also confirmed that decay rates vary dramatically with temperature. We obtained surface temperatures of approximately $-50\,^\circ$C with a simple cooling technique: discharging inverted cans of "canned air" duster spray directly onto the chips. At these temperatures, we typically found that fewer than 1% of bits decayed even after 10 minutes without power. To test the limits of this effect, we submerged DRAM modules in liquid nitrogen (ca. $-196\,^\circ$C) and saw decay of only 0.17% after 60 minutes out of the computer.

In Section 4, we present several attacks that exploit DRAM remanence to acquire memory images from which keys and other sensitive data can be extracted. Our attacks come in three variants, of increasing resistance to countermeasures. The simplest is to reboot the machine and launch a custom kernel with a small memory footprint that gives the adversary access to the retained memory. A more advanced attack briefly cuts power to the machine, then restores power and boots a custom kernel; this deprives the operating system of any opportunity to scrub memory before shutting down. An even stronger attack cuts the power and then transplants the DRAM modules to a second PC prepared by the attacker, which extracts their state. This attack additionally deprives the original BIOS and PC hardware of any chance to clear the memory on boot. We have implemented imaging kernels for use with network booting or a USB drive.

If the attacker is forced to cut power to the memory for too long, the data will become corrupted. We propose three methods for reducing corruption and for correcting errors in recovered encryption keys. The first is to cool the memory chips prior to cutting power, which dramatically reduces the error rate. The second is to apply algorithms we have developed for correcting errors in private and symmetric keys. The third is to replicate the physical conditions under which the data was recovered and experimentally measure the decay properties of each memory location; with this information, the attacker can conduct an accelerated error correction procedure. These techniques can be used alone or in combination.

In Section 5, we explore the second error correction method: novel algorithms that can reconstruct cryptographic keys even with relatively high bit-error rates. Rather than attacking the key directly, our methods consider values derived from it, such as key schedules, that provide a higher degree of redundancy. For performance reasons, many applications precompute these values and keep them in memory for as long as the key itself is in use. To reconstruct an AES key, for example, we treat the decayed key schedule as an error correcting code and find the most likely values for the original key. Applying this method to keys with 10% of bits decayed, we can reconstruct nearly any 128-bit AES key within a few seconds. We have devised reconstruction techniques for AES, DES, and RSA keys, and we expect that similar approaches will be possible for other cryptosystems. The vulnerability of precomputation products to such attacks suggests an interesting trade-off between efficiency and security. In Section 6, we present fully automatic techniques for identifying such keys from memory images, even in the presence of bit errors.

We demonstrate the effectiveness of these attacks in Section 7 by attacking several widely used disk encryption products, including BitLocker, TrueCrypt, and FileVault. We have developed a fully automated demonstration attack against BitLocker that allows access to the contents of the disk with only a few minutes of computation. Notably, using BitLocker with a Trusted Platform Module (TPM) sometimes makes it *less* secure, allowing an attacker to gain access to the data even if the machine is stolen while it is completely powered off.

It may be difficult to prevent all the attacks that we describe even with significant changes to the way encryption products are designed and used, but in practice there are a number of safeguards that can provide partial resistance. In Section 8, we suggest a variety of mitigation strategies ranging from methods that average users can apply today to long-term software and hardware changes. Each remedy has limitations and trade-offs. As we conclude in Section 9, it seems there is no simple fix for DRAM remanence vulnerabilities.

**Online resources** A video demonstration of our attacks and source code for some of our tools are available at http://citp.princeton.edu/memory.

## 2 Previous Work

Previous researchers have suggested that data in DRAM might survive reboots, and that this fact might have security implications. To our knowledge, however, ours is the first security study to focus on this phenomenon, the first to consider how to reconstruct symmetric keys in the presence of errors, the first to apply such attacks to real disk encryption systems, and the first to offer a systematic discussion of countermeasures.

We owe the suggestion that modern DRAM contents can survive cold boot to Pettersson [33], who seems to have obtained it from Chow, Pfaff, Garfinkel, and Rosenblum [13]. Pettersson suggested that remanence across cold boot could be used to acquire forensic memory images and obtain cryptographic keys, although he did not experiment with the possibility. Chow *et al.* discovered this property in the course of an experiment on data lifetime in running systems. While they did not exploit the

| | Memory Type | Chip Maker | Memory Density | Make/Model | Year |
|---|---|---|---|---|---|
| A | SDRAM | Infineon | 128Mb | Dell Dimension 4100 | 1999 |
| B | DDR | Samsung | 512Mb | Toshiba Portégé | 2001 |
| C | DDR | Micron | 256Mb | Dell Inspiron 5100 | 2003 |
| D | DDR2 | Infineon | 512Mb | IBM T43p | 2006 |
| E | DDR2 | Elpida | 512Mb | IBM x60 | 2007 |
| F | DDR2 | Samsung | 512Mb | Lenovo 3000 N100 | 2007 |

Table 1: Test systems we used in our experiments

property, they remark on the negative security implications of relying on a reboot to clear memory.

In a recent presentation, MacIver [31] stated that Microsoft considered memory remanence attacks in designing its BitLocker disk encryption system. He acknowledged that BitLocker is vulnerable to having keys extracted by cold-booting a machine when it is used in "basic mode" (where the encrypted disk is mounted automatically without requiring a user to enter any secrets), but he asserted that BitLocker is not vulnerable in "advanced modes" (where a user must provide key material to access the volume). He also discussed cooling memory with dry ice to extend the retention time. MacIver apparently has not published on this subject.

It has been known since the 1970s that DRAM cell contents survive to some extent even at room temperature and that retention times can be increased by cooling. In a 1978 experiment [29], a DRAM showed no data loss for a full week without refresh when cooled with liquid nitrogen. Anderson [2] briefly discusses remanence in his 2001 book:

> [A]n attacker can … exploit … memory remanence, the fact that many kinds of computer memory retain some trace of data that have been stored there. … [M]odern RAM chips exhibit a wide variety of memory remanence behaviors, with the worst of them keeping data for several seconds even at room temperature…

Anderson cites Skorobogatov [40], who found significant data retention times with *static* RAMs at room temperature. Our results for modern DRAMs show even longer retention in some cases.

Anderson's main focus is on "burn-in" effects that occur when data is stored in RAM for an extended period. Gutmann [22, 23] also examines "burn-in," which he attributes to physical changes that occur in semiconductor memories when the same value is stored in a cell for a long time. Accordingly, Gutmann suggests that keys should not be stored in one memory location for longer than several minutes. Our findings concern a different phenomenon: the remanence effects we have studied occur in modern DRAMs even when data is stored only momentarily. These effects do not result from the kind of physical changes that Gutmann described, but rather from the capacitance of DRAM cells.

Other methods for obtaining memory images from live systems include using privileged software running under the host operating system [43], or using DMA transfer on an external bus [19], such as PCI [12], mini-PCI, Firewire [8, 15, 16], or PC Card. Unlike these techniques, our attacks do not require access to a privileged account on the target system, they do not require specialized hardware, and they are resistant to operating system countermeasures.

## 3 Characterizing Remanence Effects

A DRAM cell is essentially a capacitor. Each cell encodes a single bit by either charging or not charging one of the capacitor's conductors. The other conductor is hard-wired either to power or to ground, depending on the cell's address within the chip [37, 23].

Over time, charge will leak out of the capacitor, and the cell will lose its state or, more precisely, it will decay to its *ground state*, either zero or one depending on whether the fixed conductor of the capacitor is hard-wired to ground or power. To forestall this decay, the cell must be *refreshed*, meaning that the capacitor must be re-charged to hold its value. Specifications for DRAM chips give a *refresh time*, which is the maximum interval that is supposed to pass before a cell is refreshed. The standard refresh time (usually on the order of milliseconds) is meant to achieve extremely high reliability for normal computer operations where even infrequent bit errors could cause serious problems; however, a failure to refresh any individual DRAM cell within this time has only a tiny probability of actually destroying the cell's contents.

We conducted a series of experiments to characterize DRAM remanence effects and better understand the security properties of modern memories. We performed trials using PC systems with different memory technologies, as shown in Table 1. These systems included models from several manufacturers and ranged in age from 9 years to 6 months.

## 3.1 Decay at operating temperature

Using a modified version of our PXE memory imaging program (see Section 4.1), we filled representative memory regions with a pseudorandom pattern. We read back these memory regions after varying periods of time without refresh and under different temperature conditions, and measured the error rate of each sample. The error rate is the number of bit errors in each sample (the Hamming distance from the pattern we had written) divided by the total number of bits we measured. Since our pseudorandom test pattern contained roughly equal numbers of zeros and ones, we would expect fully decayed memory to have an error rate of approximately 50% .

Our first tests measured the decay rate of each memory module under normal operating temperature, which ranged from 25.5 °C to 44.1 °C, depending on the machine (see Figures 1, 2, and 3). We found that the dimensions of the decay curves varied considerably between machines, with the fastest exhibiting complete data loss in approximately 2.5 seconds and the slowest taking an average of 35 seconds. However, the decay curves all display a similar shape, with an initial period of slow decay, followed by an intermediate period of rapid decay, and then a final period of slow decay.

We calculated best fit curves to the data using the logistic function because MOSFETs, the basic components of a DRAM cell, exhibit a logistic decay curve. We found that machines using newer memory technologies tend to exhibit a shorter time to total decay than machines using older memory technologies, but even the shorter times are long enough to facilitate most of our attacks. We ascribe this trend to the increasing density of the DRAM cells as the technology improves; in general, memory with higher densities have a shorter window where data is recoverable. While this trend might make DRAM retention attacks more difficult in the future, manufacturers also generally seek to *increase* retention times, because DRAMs with long retention require less frequent refresh and have lower power consumption.

## 3.2 Decay at reduced temperature

It has long been known that low temperatures can significantly increase memory devices' retention times [29, 2, 46, 23, 41, 40]. To measure this effect, we performed a second series of tests using machines A–D.

In each trial, we loaded a pseudorandom test pattern into memory, and, with the computer running, cooled the memory module to approximately −50 °C. We then powered off the machine and maintained this temperature until power was restored. We achieved these temperatures using commonly available "canned air" duster products (see Section 4.2), which we discharged, with the can inverted, directly onto the chips.

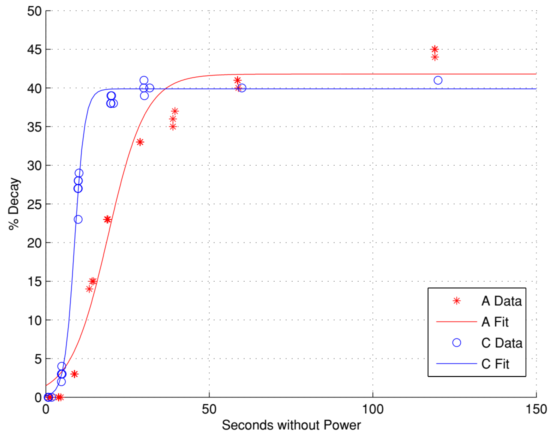


Figure 1: Machines A and C

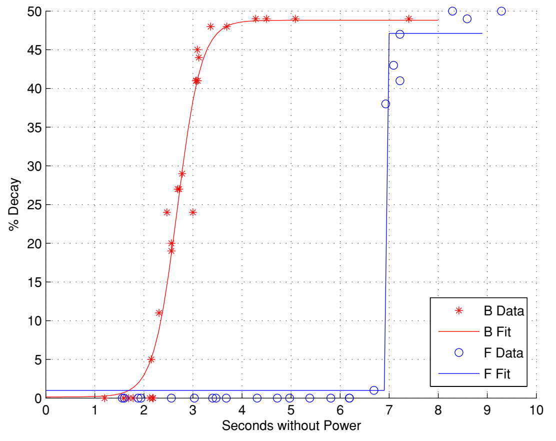


Figure 2: Machines B and F

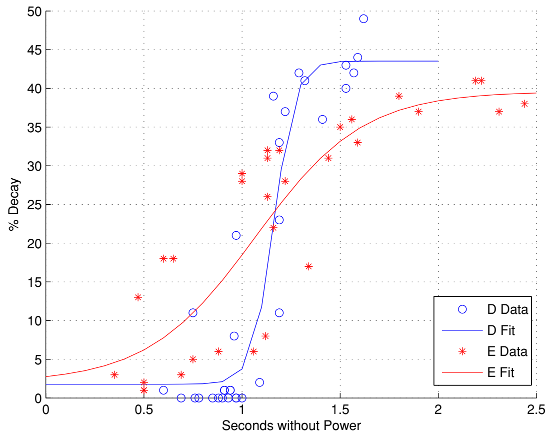


Figure 3: Machines D and E

|  | Seconds w/o power | Error % at operating temp. | Error % at $-50\,^{\circ}$C |
|---|---|---|---|
| A | 60 | 41 | (no errors) |
|  | 300 | 50 | 0.000095 |
| B | 360 | 50 | (no errors) |
|  | 600 | 50 | 0.000036 |
| C | 120 | 41 | 0.00105 |
|  | 360 | 42 | 0.00144 |
| D | 40 | 50 | 0.025 |
|  | 80 | 50 | 0.18 |

Table 2: Effect of cooling on error rates

As expected, we observed a significantly lower rate of decay under these reduced temperatures (see Table 2). On all of our sample DRAMs, the decay rates were low enough that an attacker who cut power for 60 seconds would recover 99.9% of bits correctly.

As an extreme test of memory cooling, we performed another experiment using liquid nitrogen as an additional cooling agent. We first cooled the memory module of Machine A to $-50\,^{\circ}$C using the "canned air" product. We then cut power to the machine, and quickly removed the DRAM module and placed it in a canister of liquid nitrogen. We kept the memory module submerged in the liquid nitrogen for 60 minutes, then returned it to the machine. We measured only 14,000 bit errors within a 1 MB test region (0.17% decay). This suggests that, even in modern memory modules, data may be recoverable for hours or days with sufficient cooling.

## 3.3 Decay patterns and predictability

We observed that the DRAMs we studied tended to decay in highly nonuniform patterns. While these patterns varied from chip to chip, they were very predictable in most of the systems we tested. Figure 4 shows the decay in one memory region from Machine A after progressively longer intervals without power.

There seem to be several components to the decay patterns. The most prominent is a gradual decay to the "ground state" as charge leaks out of the memory cells. In the DRAM shown in Figure 4, blocks of cells alternate between a ground state of 0 and a ground state of 1, resulting in the series of horizontal bars. Other DRAM models and other regions within this DRAM exhibited different ground states, depending on how the cells are wired.

We observed a small number of cells that deviated from the "ground state" pattern, possibly due to manufacturing variation. In experiments with 20 or 40 runs, a few "retrograde" cells (typically $\sim 0.05\%$ of memory cells, but larger in a few devices) always decayed to the opposite value of the one predicted by the surrounding ground state pattern. An even smaller number of cells decayed in different directions across runs, with varying probabilities.

Apart from their eventual states, the *order* in which different cells decayed also appeared to be highly predictable. At a fixed temperature, each cell seems to decay after a consistent length of time without power. The relative order in which the cells decayed was largely fixed, even as the decay times were changed by varying the temperature. This may also be a result of manufacturing variations, which result in some cells leaking charge faster than others.

To visualize this effect, we captured degraded memory images, including those shown in Figure 4, after cutting power for intervals ranging from 1 second to 5 minutes, in 1 second increments. We combined the results into a video (available on our web site). Each test interval began with the original image freshly loaded into memory. We might have expected to see a large amount of variation between frames, but instead, most bits appear stable from frame to frame, switching values only once, after the cell's decay interval. The video also shows that the decay intervals themselves follow higher order patterns, likely related to the physical geometry of the DRAM.

## 3.4 BIOS footprints and memory wiping

Even if memory contents remain intact while power is off, the system BIOS may overwrite portions of memory when the machine boots. In the systems we tested, the BIOS overwrote only relatively small fractions of memory with its own code and data, typically a few megabytes concentrated around the bottom of the address space.

On many machines, the BIOS can perform a destructive memory check during its Power-On Self Test (POST). Most of the machines we examined allowed this test to be disabled or bypassed (sometimes by enabling an option called "Quick Boot").

On other machines, mainly high-end desktops and servers that support ECC memory, we found that the BIOS cleared memory contents without any override option. ECC memory must be set to a known state to avoid spurious errors if memory is read without being initialized [6], and we believe many ECC-capable systems perform this wiping operation whether or not ECC memory is installed.

ECC DRAMs are not immune to retention effects, and an attacker could transfer them to a non-ECC machine that does not wipe its memory on boot. Indeed, ECC memory could turn out to *help* the attacker by making DRAM more resistant to bit errors.

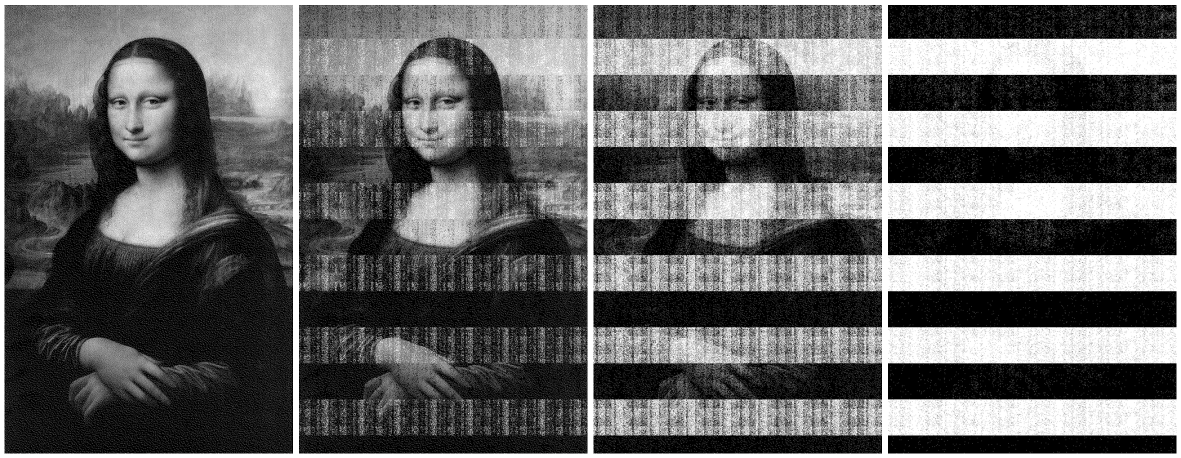


Figure 4: We loaded a bitmap image into memory on Machine A, then cut power for varying lengths of time. After 5 seconds (left), the image is indistinguishable from the original. It gradually becomes more degraded, as shown after 30 seconds, 60 seconds, and 5 minutes.

## 4 Imaging Residual Memory

Imaging residual memory contents requires no special equipment. When the system boots, the memory controller begins refreshing the DRAM, reading and rewriting each bit value. At this point, the values are fixed, decay halts, and programs running on the system can read any data present using normal memory-access instructions.

### 4.1 Imaging tools

One challenge is that booting the system will necessarily overwrite some portions of memory. Loading a full operating system would be very destructive. Our approach is to use tiny special-purpose programs that, when booted from either a warm or cold reset state, produce accurate dumps of memory contents to some external medium. These programs use only trivial amounts of RAM, and their memory offsets used can be adjusted to some extent to ensure that data structures of interest are unaffected.

Our memory-imaging tools make use of several different attack vectors to boot a system and extract the contents of its memory. For simplicity, each saves memory images to the medium from which it was booted.

**PXE network boot** Most modern PCs support network booting via Intel's Preboot Execution Environment (PXE) [25], which provides rudimentary startup and network services. We implemented a tiny (9 KB) standalone application that can be booted via PXE and whose only function is streaming the contents of system RAM via a UDP-based protocol. Since PXE provides a universal API for accessing the underlying network hardware, the same binary image will work unmodified on any PC system with PXE support. In a typical attack setup, a laptop connected to the target machine via an Ethernet crossover cable runs DHCP and TFTP servers as well as a simple client application for receiving the memory data. We have extracted memory images at rates up to 300 Mb/s (around 30 seconds for a 1 GB RAM) with gigabit Ethernet cards.

**USB drives** Alternatively, most PCs can boot from an external USB device such as a USB hard drive or flash device. We implemented a small (10 KB) plug-in for the SYSLINUX bootloader [3] that can be booted from an external USB device or a regular hard disk. It saves the contents of system RAM into a designated data partition on this device. We succeeded in dumping 1 GB of RAM to a flash drive in approximately 4 minutes.

**EFI boot** Some recent computers, including all Intel-based Macintosh computers, implement the Extensible Firmware Interface (EFI) instead of a PC BIOS. We have also implemented a memory dumper as an EFI netboot application. We have achieved memory extraction speeds up to 136 Mb/s, and we expect it will be possible to increase this throughput with further optimizations.

**iPods** We have installed memory imaging tools on an Apple iPod so that it can be used to covertly capture memory dumps without impacting its functionality as a music player. This provides a plausible way to conceal the attack in the wild.

### 4.2 Imaging attacks

An attacker could use imaging tools like ours in a number of ways, depending on his level of access to the system and the countermeasures employed by hardware and software.

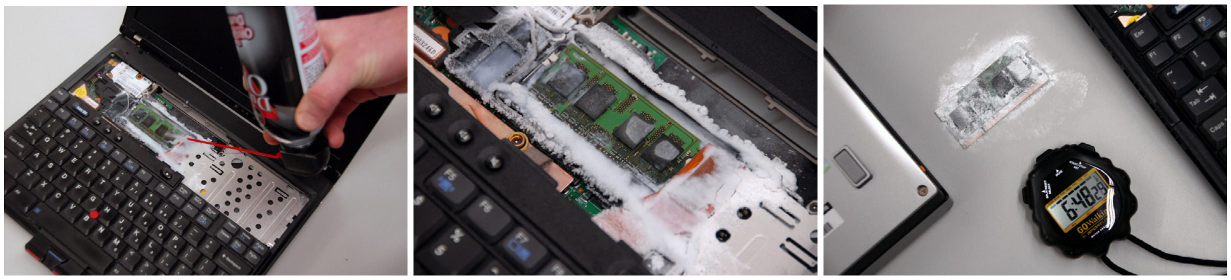


Figure 5: Before powering off the computer, we spray an upside-down canister of multipurpose duster directly onto the memory chips, cooling them to $-50\,^{\circ}\mathrm{C}$. At this temperature, the data will persist for several minutes after power loss with minimal error, even if we remove the DIMM from the computer.

**Simple reboots** The simplest attack is to reboot the machine and configure the BIOS to boot the imaging tool. A warm boot, invoked with the operating system's restart procedure, will normally ensure that the memory has no chance to decay, though software will have an opportunity to wipe sensitive data prior to shutdown. A cold boot, initiated using the system's restart switch or by briefly removing and restoring power, will result in little or no decay depending on the memory's retention time. Restarting the system in this way denies the operating system and applications any chance to scrub memory before shutting down.

**Transferring DRAM modules** Even if an attacker cannot force a target system to boot memory-imaging tools, or if the target employs countermeasures that erase memory contents during boot, DIMM modules can be physically removed and their contents imaged using another computer selected by the attacker.

Some memory modules exhibit far faster decay than others, but as we discuss in Section 3.2 above, cooling a module before powering it off can slow decay sufficiently to allow it to be transferred to another machine with minimal decay. Widely-available "canned air" dusters, usually containing a compressed fluorohydrocarbon refrigerant, can easily be used for this purpose. When the can is discharged in an inverted position, as shown in Figure 5, it dispenses its contents in liquid form instead of as a gas. The rapid drop in pressure inside the can lowers the temperature of the discharge, and the subsequent evaporation of the refrigerant causes a further chilling. By spraying the contents directly onto memory chips, we can cool their surfaces to $-50\,^{\circ}\mathrm{C}$ and below. If the DRAM is cooled to this temperature before power is cut and kept cold, we can achieve nearly lossless data recovery even after the chip is out of the computer for several minutes.

Removing the memory modules can also allow the attacker to image memory in address regions where standards BIOSes load their own code during boot. The attacker could remove the primary memory module from the target machine and place it into the secondary DIMM slot (in the same machine or another machine), effectively remapping the data to be imaged into a different part of the address space.

## 5 Key Reconstruction

Our experiments (see Section 3) show that it is possible to recover memory contents with few bit errors even after cutting power to the system for a brief time, but the presence of even a small amount of error complicates the process of extracting correct cryptographic keys. In this section we present algorithms for correcting errors in symmetric and private keys. These algorithms can correct most errors quickly even in the presence of relatively high bit error probabilities in the range of 5% to 50%, depending on the type of key.

A naïve approach to key error correction is to brute-force search over keys with a low Hamming distance from the decayed key that was retrieved from memory, but this is computationally burdensome even with a moderate amount of unidirectional error. As an example, if only 10% of the ones have decayed to zeros in our memory image, the data recovered from a 256-bit key with an equal number of ones and zeroes has an expected Hamming distance of 12 from the actual key, and the number of such keys is $\binom{128+12}{12} > 2^{56}$.

Our algorithms achieve significantly better performance by considering data other than the raw form of the key. Most encryption programs speed up computation by storing data precomputed from the encryption keys—for block ciphers, this is most often a key schedule, with subkeys for each round; for RSA, this is an extended form of the private key which includes the primes $p$ and $q$ and several other values derived from $d$. This data contains much more structure than the key itself, and we can use this structure to efficiently reconstruct the original key even in the presence of errors.

These results imply an interesting trade-off between efficiency and security. All of the disk encryption systems we studied (see Section 7) precompute key schedules and keep them in memory for as long as the encrypted disk is mounted. While this practice saves some computation for each disk block that needs to be encrypted or decrypted, we find that it greatly simplifies key recovery attacks.

Our approach to key reconstruction has the advantage that it is completely self-contained, in that we can recover the key without having to test the decryption of ciphertext. The data derived from the key, and not the decoded plaintext, provides a certificate of the likelihood that we have found the correct key.

We have found it useful to adopt terminology from coding theory. We may imagine that the expanded key schedule forms a sort of *error correcting code* for the key, and the problem of reconstructing a key from memory may be recast as the problem of finding the closest *code word* (valid key schedule) to the data once it has been passed through a channel that has introduced bit errors.

**Modeling the decay** Our experiments showed that almost all memory bits tend to decay to predictable ground states, with only a tiny fraction flipping in the opposite direction. In describing our algorithms, we assume, for simplicity, that all bits decay to the same ground state. (They can be implemented without this requirement, assuming that the ground state of each bit is known.)

If we assume we have no knowledge of the decay patterns other than the ground state, we can model the decay with the *binary asymmetric channel*, in which the probability of a 1 flipping to 0 is some fixed $\delta_0$ and the probability of a 0 flipping to a 1 is some fixed $\delta_1$.

In practice, the probability of decaying to the ground state approaches 1 as time goes on, while the probability of flipping in the opposite direction remains relatively constant and tiny (less than 0.1% in our tests). The ground state decay probability can be approximated from recovered key data by counting the fraction of 1s and 0s, assuming that the original key data contained roughly equal proportions of each value.

We also observed that bits tended to decay in a predictable order that could be learned over a series of timed decay trials, although the actual order of decay appeared fairly random with respect to location. An attacker with the time and physical access to run such a series of tests could easily adapt any of the approaches in this section to take this order into account and improve the performance of the error-correction. Ideally such tests would be able to replicate the conditions of the memory extraction exactly, but knowledge of the decay order combined with an estimate of the fraction of bit flips is enough to give a very good estimate of an individual decay probability of each bit. This probability can be used in our reconstruction algorithms to prioritize guesses.

For simplicity and generality, we will analyze the algorithms assuming no knowledge of this decay order.

## 5.1 Reconstructing DES keys

We first apply these methods to develop an error correction technique for DES. The DES key schedule algorithm produces 16 subkeys, each a permutation of a 48-bit subset of bits from the original 56-bit key. Every bit from the original key is repeated in about 14 of the 16 subkeys.

In coding theory terms, we can treat the DES key schedule as a repetition code: the message is a single bit, and the corresponding codeword is a sequence of $n$ copies of this bit. If $\delta_0 = \delta_1 < \frac{1}{2}$, the optimal decoding of such an $n$-bit codeword is 0 if more than $n/2$ of the recovered bits are 0, and 1 otherwise. For $\delta_0 \neq \delta_1$, the optimal decoding is 0 if more than $nr$ of the recovered bits are 0 and 1 otherwise, where

$$r = \frac{\log(1-\delta_0) - \log\delta_1}{\log(1-\delta_0) + \log(1-\delta_1) - \log\delta_1 - \log\delta_0}.$$

For $\delta_0 = .1$ and $\delta_1 = .001$ (that is, we are in a block with ground state 0), $r = .75$ and this approach will fail to correctly decode a bit only if more than 3 of the 14 copies of a 0 decay to a 1, or more than 11 of the 14 copies of a 1 decay to 0. The probability of this event is less than $10^{-9}$. Applying the union bound, the probability that any of the 56 key bits will be incorrectly decoded is at most $56 \times 10^{-9} < 6 \times 10^{-8}$; even at 50% error, the probability that the key can be correctly decoded without resorting to brute force search is more than 98%.

This technique can be trivially extended to correct errors in Triple DES keys. Since Triple DES applies the same key schedule algorithm to two or three 56-bit key components (depending on the version of Triple DES), the probability of correctly decoding each key bit is the same as for regular DES. With a decay rate of $\delta_0 = .5$ and probability $\delta_1 = .001$ of bit flips in the opposite direction, we can correctly decode a 112-bit Triple DES key with at least 97% probability and a 168-bit key with at least 96% probability.

## 5.2 Reconstructing AES keys

The AES key schedule has a more complex structure than the DES key schedule, but we can still use it to efficiently reconstruct a key in the presence of errors.

A seemingly reasonable approach to this problem would be to search keys in order of distance to the recovered key and output any key whose schedule is sufficiently close to the recovered schedule. Our implementation of this algorithm took twenty minutes to search $10^9$ candidate keys in order to reconstruct a key in which 7 zeros

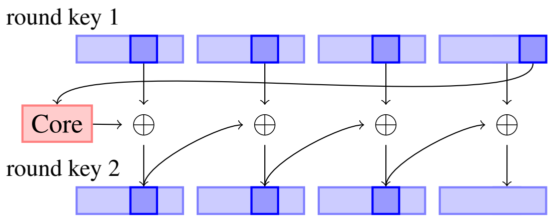


Figure 6: In the 128-bit AES key schedule, three bytes of each round key are entirely determined by four bytes of the preceding round key.

had flipped to ones. At this rate it would take ten days to reconstruct a key with 11 bits flipped.

We can do significantly better by taking advantage of the structure of the AES key schedule. Instead of trying to correct an entire key at once, we can examine a smaller set of bytes at a time. The high amount of linearity in the key schedule is what permits this separability—we can take advantage of pieces that are small enough to brute force optimal decodings for, yet large enough that these decodings are useful to reconstruct the overall key. Once we have a list of possible decodings for these smaller pieces of the key in order of likelihood, we can combine them into a full key to check against the key schedule.

Since each of the decoding steps is quite fast, the running time of the entire algorithm is ultimately limited by the number of combinations we need to check. The number of combinations is still roughly exponential in the number of errors, but it is a vast improvement over brute force searching and is practical in many realistic cases.

**Overview of the algorithm** For 128-bit keys, an AES key expansion consists of 11 four-word (128-bit) round keys. The first round key is equal to the key itself. Each remaining word of the key schedule is generated either by XORing two words of the key schedule together, or by performing the key schedule core (in which the bytes of a word are rotated and each byte is mapped to a new value) on a word of the key schedule and XORing the result with another word of the key schedule.

Consider a "slice" of the first two round keys consisting of byte $i$ from words 1-3 of the first two round keys, and byte $i-1$ from word 4 of the first round key (as shown in Figure 6). This slice is 7 bytes long, but is uniquely determined by the four bytes from the key. In theory, there are still $2^{32}$ possibilities to examine for each slice, but we can do quite well by examining them in order of distance to the recovered key. For each possible set of 4 key bytes, we generate the relevant three bytes of the next round key and calculate the probability, given estimates of $\delta_0$ and $\delta_1$, that these seven bytes might have decayed to the corresponding bytes of the recovered round keys.

Now we proceed to guess candidate keys, where a candidate contains a value for each slice of bytes. We consider the candidates in order of decreasing total likelihood as calculated above. For each candidate key we consider, we calculate the expanded key schedule and ask if the likelihood of that expanded key schedule decaying to our recovered key schedule is sufficiently high. If so, then we output the corresponding key as a guess.

When one of $\delta_0$ or $\delta_1$ is very small, this algorithm will almost certainly output a unique guess for the key. To see this, observe that a single bit flipped in the key results in a cascade of bit flips through the key schedule, half of which are likely to flip in the "wrong" direction.

Our implementation of this algorithm is able to reconstruct keys with 15% error (that is, $\delta_0 = .15$ and $\delta_1 = .001$) in a fraction of a second, and about half of keys with 30% error within 30 seconds.

This idea can be extended to 256-bit keys by dividing the words of the key into two sections—words 1–3 and 8, and words 4–7, for example—then comparing the words of the third and fourth round keys generated by the bytes of these words and combining the result into candidate round keys to check.

## 5.3 Reconstructing tweak keys

The same methods can be applied to reconstruct keys for tweakable encryption modes [30], which are commonly used in disk encryption systems.

**LRW** LRW augments a block cipher $E$ (and key $K_1$) by computing a "tweak" $X$ for each data block and encrypting the block using the formula $E_{K_1}(P \oplus X) \oplus X$. A tweak key $K_2$ is used to compute the tweak, $X = K_2 \otimes I$, where $I$ is the logical block identifier. The operations $\oplus$ and $\otimes$ are performed in the finite field $GF(2^{128})$.

In order to speed tweak computations, implementations commonly precompute multiplication tables of the values $K_2 x^i \mod P$, where $x$ is the primitive element and $P$ is an irreducible polynomial over $GF(2^{128})$ [26]. In practice, $Qx \mod P$ is computed by shifting the bits of $Q$ left by one and possibly XORing with $P$.

Given a value $K_2 x^i$, we can recover nearly all of the bits of $K_2$ simply by shifting right by $i$. The number of bits lost depends on $i$ and the nonzero elements of $P$. An entire multiplication table will contain many copies of nearly all of the bits of $K_2$, allowing us to reconstruct the key in much the same way as the DES key schedule.

As an example, we apply this method to reconstruct the LRW key used by the TrueCrypt 4 disk encryption system. TrueCrypt 4 precomputes a 4048-byte multiplication table consisting of 16 blocks of 16 lines of 4 words of 4 bytes each. Line 0 of block 14 contains the tweak key.

The multiplication table is generated line by line from the LRW key by iteratively applying the shift-and-XOR multiply function to generate four new values, and then XORing all combinations of these four values to create 16 more lines of the table. The shift-and-XOR operation is performed 64 times to generate the table, using the irreducible polynomial $P = x^{128} + x^7 + x^2 + x + 1$. For any of these 64 values, we can shift right $i$ times to recover $128 - (8 + i)$ of the bits of $K_2$, and use these recovered values to reconstruct $K_2$ with high probability.

**XEX and XTS** For XEX [35] and XTS [24] modes, the tweak for block $j$ in sector $I$ is $X = E_{K_2}(I) \otimes x^j$, where $I$ is encrypted with AES and $x$ is the primitive element of $GF(2^{128})$. Assuming the key schedule for $K_2$ is kept in memory, we can use the AES key reconstruction techniques to reconstruct the tweak key.

### 5.4 Reconstructing RSA private keys

An RSA public key consists of the modulus $N$ and the public exponent $e$, while the private key consists of the private exponent $d$ and several optional values: prime factors $p$ and $q$ of $N$, $d \mod (p-1)$, $d \mod (q-1)$, and $q^{-1} \mod p$. Given $N$ and $e$, any of the private values is sufficient to generate the others using the Chinese remainder theorem and greatest common divisor algorithms. In practice, RSA implementations store some or all optional values to speed up computation.

There have been a number of results on efficiently reconstructing RSA private keys given a fraction of the bits of private key data. Let $n = \lg N$. $N$ can be factored in polynomial time given the $n/4$ least significant bits of $p$ (Coppersmith [14]), given the $n/4$ least significant bits of $d$ (Boneh, Durfee, and Frankel [9]), or given the $n/4$ least significant bits of $d \mod (p-1)$ (Blömer and May [7]).

These previous results are all based on Coppersmith's method of finding bounded solutions to polynomial equations using lattice basis reduction; the number of contiguous bits recovered from the most or least significant bits of the private key data determines the additive error tolerated in the solution. In our case, the errors may be distributed across all bits of the key data, so we are searching for solutions with low Hamming weight, and these previous approaches do not seem to be directly applicable.

Given the public modulus $N$ and the values $\tilde{p}$ and $\tilde{q}$ recovered from memory, we can deduce values for the original $p$ and $q$ by iteratively reconstructing them from the least-significant bits. For unidirectional decay with probability $\delta$, bits $p_i$ and $q_i$ are uniquely determined by $N_i$ and our guesses for the $i-1$ lower-order bits of $p$ and $q$ (observe that $p_0 = q_0 = 1$), except in the case when $\tilde{p}_i$ and $\tilde{q}_i$ are both in the ground state. This yields a branching process with expected degree $\frac{(3+\delta)^2}{8}$. If decay is not unidirectional, we may use the estimated probabilities to weight the branches at each bit.

Combined with a few heuristics—for example, choose the most likely state first, prune nodes by bounds on the solution, and iteratively increase the bit flips allowed—this results in a practical algorithm for reasonable error rates. This process can likely be improved substantially using additional data recovered from the private key.

We tested an implementation of the algorithm on a fast modern machine. For fifty trials with 1024-bit primes (2048-bit keys) and $\delta = 4\%$, the median reconstruction time was 4.5 seconds. The median number of nodes visited was 16,499, the mean was 621,707, and the standard deviation was 2,136,870. For $\delta = 6\%$, reconstruction required a median of 2.5 minutes, or 227,763 nodes visited.

For 512-bit primes and $\delta = 10\%$, reconstruction required a median of 1 minute, or 188,702 nodes visited.

For larger error rates, we can attempt to reconstruct only the first $n/4$ bits of the key using this process and use the lattice techniques to reconstruct the rest of the key; these computations generally take several hours in practice. For a 1024-bit RSA key, we would need to recover 256 bits of a factor. The expected depth of the tree from our branching reconstruction process would be $(\frac{1}{2} + \delta)^2 256$ (assuming an even distribution of 0s and 1s) and the expected fraction of branches that would need to be examined is $1/2 + \delta^2$.

## 6 Identifying Keys in Memory

Extracting encryption keys from memory images requires a mechanism for locating the target keys. A simple approach is to test every sequence of bytes to see whether it correctly decrypts some known plaintext. Applying this method to a 1 GB memory image known to contain a 128-bit symmetric key aligned to some 4-byte machine word implies at most $2^{28}$ possible key values. However, this is only the case if the memory image is perfectly accurate. If there are bit errors in the portion of memory containing the key, the search quickly becomes intractable.

We have developed fully automatic techniques for locating symmetric encryption keys in memory images, even in the presence of bit errors. Our approach is similar to the one we used to correct key bit errors in Section 5. We target the key schedule instead of the key itself, searching for blocks of memory that satisfy (or are close to satisfying) the combinatorial properties of a valid key schedule. Using these methods we have been able to recover keys from closed-source encryption programs without having to disassemble them and reconstruct their key data structures, and we have even recovered partial key schedules that had been overwritten by another program when the memory was reallocated.

Although previous approaches to key recovery do not require a scheduled key to be present in memory, they have other practical drawbacks that limit their usefulness for our purposes. Shamir and van Someren [39] propose visual and statistical tests of randomness which can quickly identify regions of memory that might contain key material, but their methods are prone to false positives that complicate testing on decayed memory images. Even perfect copies of memory often contain large blocks of random-looking data that might pass these tests (e.g., compressed files). Pettersson [33] suggests a plausibility test for locating a particular program data structure that contains key material based on the range of likely values for each field. This approach requires the operator to manually derive search heuristics for each encryption application, and it is not very robust to memory errors.

## 6.1 Identifying AES keys

In order to identify scheduled AES keys in a memory image, we propose the following algorithm:

1. Iterate through each byte of memory. Treat the following block of 176 or 240 bytes of memory as an AES key schedule.

2. For each word in the potential key schedule, calculate the Hamming distance from that word to the key schedule word that should have been generated from the surrounding words.

3. If the total number of bits violating the constraints on a correct AES key schedule is sufficiently small, output the key.

We created an application called `keyfind` that implements this algorithm for 128- and 256-bit AES keys. The program takes a memory image as input and outputs a list of likely keys. It assumes that key schedules are contained in contiguous regions of memory and in the byte order used in the AES specification [1]; this can be adjusted to target particular cipher implementations. A threshold parameter controls how many bit errors will be tolerated in candidate key schedules. We apply a quick test of entropy to reduce false positives.

We expect that this approach can be applied to many other ciphers. For example, to identify DES keys based on their key schedule, calculate the distance from each potential subkey to the permutation of the key. A similar method works to identify the precomputed multiplication tables used for advanced cipher modes like LRW (see Section 5.3).

## 6.2 Identifying RSA keys

Methods proposed for identifying RSA private keys range from the purely algebraic (Shamir and van Someren suggest, for example, multiplying adjacent key-sized blocks of memory [39]) to the *ad hoc* (searching for the RSA Object Identifiers found in ASN.1 key objects [34]). The former ignores the widespread use of standard key formats, and the latter seems insufficiently robust.

The most widely used format for an RSA private key is specified in PKCS #1 [36] as an ASN.1 object of type RSAPrivateKey with the following fields: version, modulus $n$, publicExponent $e$, privateExponent $d$, prime1 $p$, prime2 $q$, exponent1 $d \mod (p-1)$, exponent2 $d \mod (q-1)$, coefficient $q^{-1} \mod p$, and optional other information. This object, packaged in DER encoding, is the standard format for storage and interchange of private keys.

This format suggests two techniques we might use for identifying RSA keys in memory: we could search for known *contents* of the fields, or we could look for memory that matches the *structure* of the DER encoding. We tested both of these approaches on a computer running Apache 2.2.3 with `mod_ssl`.

One value in the key object that an attacker is likely to know is the public modulus. (In the case of a web server, the attacker can obtain this and the rest of the public key by querying the server.) We tried searching for the modulus in memory and found several matches, all of them instances of the server's public or private key.

We also tested a key finding method described by Ptacek [34] and others: searching for the RSA Object Identifiers that should mark ASN.1 key objects. This technique yielded only false positives on our test system.

Finally, we experimented with a new method, searching for identifying features of the DER-encoding itself. We looked for the sequence identifier (0x30) followed a few bytes later by the DER encoding of the RSA version number and then by the beginning of the DER encoding of the next field (02 01 00 02). This method found several copies of the server's private key, and no false positives. To locate keys in decayed memory images, we can adapt this technique to search for sequences of bytes with low Hamming distance to these markers and check that the subsequent bytes satisfy some heuristic entropy bound.

## 7 Attacking Encrypted Disks

Encrypting hard drives is an increasingly common countermeasure against data theft, and many users assume that disk encryption products will protect sensitive data even if an attacker has physical access to the machine. A California law adopted in 2002 [10] requires disclosure of possible compromises of personal information, but offers a safe harbor whenever data was "encrypted." Though the law does not include any specific technical standards, many observers have recommended the use of full-disk or file system encryption to obtain the benefit of this safe harbor. (At least 38 other states have enacted data breach notification legislation [32].) Our results below suggest that disk encryption, while valuable, is not necessarily a sufficient defense. We find that a moderately skilled attacker can circumvent many widely used disk encryption products if a laptop is stolen while it is powered on or suspended.

We have applied some of the tools developed in this paper to attack popular on-the-fly disk encryption systems. The most time-consuming parts of these tests were generally developing system-specific attacks and setting up the encrypted disks. Actually imaging memory and locating keys took only a few minutes and were almost fully automated by our tools. We expect that most disk encryption systems are vulnerable to such attacks.

**BitLocker** BitLocker, which is included with some versions of Windows Vista, operates as a filter driver that resides between the file system and the disk driver, encrypting and decrypting individual sectors on demand. The keys used to encrypt the disk reside in RAM, in scheduled form, for as long as the disk is mounted.

In a paper released by Microsoft, Ferguson [21] describes BitLocker in enough detail both to discover the roles of the various keys and to program an independent implementation of the BitLocker encryption algorithm without reverse engineering any software. BitLocker uses the same pair of AES keys to encrypt every sector on the disk: a sector pad key and a CBC encryption key. These keys are, in turn, indirectly encrypted by the disk's master key. To encrypt a sector, the plaintext is first XORed with a pad generated by encrypting the byte offset of the sector under the sector pad key. Next, the data is fed through two diffuser functions, which use a Microsoft-developed algorithm called Elephant. The purpose of these un-keyed functions is solely to increase the probability that modifications to any bits of the ciphertext will cause unpredictable modifications to the entire plaintext sector. Finally, the data is encrypted using AES in CBC mode using the CBC encryption key. The initialization vector is computed by encrypting the byte offset of the sector under the CBC encryption key.

We have created a fully-automated demonstration attack called BitUnlocker. It consists of an external USB hard disk containing Linux, a custom SYSLINUX-based bootloader, and a FUSD [20] filter driver that allows BitLocker volumes to be mounted under Linux. To use BitUnlocker, one first cuts the power to a running Windows Vista system, connects the USB disk, and then reboots the system off of the external drive. BitUnlocker then automatically dumps the memory image to the external disk, runs `keyfind` on the image to determine candidate keys, tries all combinations of the candidates (for the sector pad key and the CBC encryption key), and, if the correct keys are found, mounts the BitLocker encrypted volume. Once the encrypted volume has been mounted, one can browse it like any other volume in Linux. On a modern laptop with 2 GB of RAM, we found that this entire process took approximately 25 minutes.

BitLocker differs from other disk encryption products in the way that it protects the keys when the disk is not mounted. In its default "basic mode," BitLocker protects the disk's master key solely with the Trusted Platform Module (TPM) found on many modern PCs. This configuration, which may be quite widely used [21], is particularly vulnerable to our attack, because the disk encryption keys can be extracted with our attacks even if the computer is powered off for a long time. When the machine boots, the keys will be loaded into RAM automatically (before the login screen) without the entry of any secrets.

It appears that Microsoft is aware of this problem [31] and recommends configuring BitLocker in "advanced mode," where it protects the disk key using the TPM along with a password or a key on a removable USB device. However, even with these measures, BitLocker is vulnerable if an attacker gets to the system while the screen is locked or the computer is asleep (though not if it is hibernating or powered off).

**FileVault** Apple's FileVault disk encryption software has been examined and reverse-engineered in some detail [44]. In Mac OS X 10.4, FileVault uses 128-bit AES in CBC mode. A user-supplied password decrypts a header that contains both the AES key and a second key $k_2$ used to compute IVs. The IV for a disk block with logical index $I$ is computed as HMAC-SHA1$_{k_2}(I)$.

We used our EFI memory imaging program to extract a memory image from an Intel-based Macintosh system with a FileVault volume mounted. Our `keyfind` program automatically identified the FileVault AES key, which did not contain any bit errors in our tests.

With the recovered AES key but not the IV key, we can decrypt 4080 bytes of each 4096 byte disk block (all except the first AES block). The IV key is present in memory. Assuming no bits in the IV key decay, an attacker can identify it by testing all 160-bit substrings of memory to see whether they create a plausible plaintext when XORed with the decryption of the first part of the disk block. The AES and IV keys together allow full decryption of the volume using programs like `vilefault` [45].

In the process of testing FileVault, we discovered that Mac OS X 10.4 and 10.5 keep multiple copies of the user's login password in memory, where they are vulnerable to imaging attacks. Login passwords are often used to protect the default keychain, which may protect passphrases for FileVault disk images.

**TrueCrypt** TrueCrypt is a popular open-source disk encryption product for the Windows, Mac OS, and Linux platforms. It supports a variety of ciphers, including AES, Serpent, and Twofish. In version 4, all ciphers used LRW mode; in version 5, they use XTS mode (see Section 5.3).

TrueCrypt stores a cipher key and a tweak key in the volume header for each disk, which is then encrypted with a separate key derived from a user-entered password.

We tested TrueCrypt versions 4.3a and 5.0a running on a Linux system. We mounted a volume encrypted with a 256-bit AES key, then briefly cut power to the system and used our memory imaging tools to record an image of the retained memory data. In both cases, our `keyfind` program was able to identify the 256-bit AES encryption key, which did not contain any bit errors. For TrueCrypt 5.0a, `keyfind` was also able to recover the 256-bit AES XTS tweak key without errors.

To decrypt TrueCrypt 4 disks, we also need the LRW tweak key. We observed that TrueCrypt 4 stores the LRW key in the four words immediately preceding the AES key schedule. In our test memory image, the LRW key did not contain any bit errors. (Had errors occurred, we could have recovered the correct key by applying the techniques we developed in Section 5.3.)

**dm-crypt** Linux kernels starting with 2.6 include built-in support for dm-crypt, an on-the-fly disk encryption subsystem. The dm-crypt subsystem handles a variety of ciphers and modes, but defaults to 128-bit AES in CBC mode with non-keyed IVs.

We tested a dm-crypt volume created and mounted using the LUKS (Linux Unified Key Setup) branch of the `cryptsetup` utility and kernel version 2.6.20. The volume used the default AES-CBC format. We briefly powered down the system and captured a memory image with our PXE kernel. Our `keyfind` program identified the correct 128-bit AES key, which did not contain any bit errors. After recovering this key, an attacker could decrypt and mount the dm-crypt volume by modifying the `cryptsetup` program to allow input of the raw key.

**Loop-AES** Loop-AES is an on-the-fly disk encryption package for Linux systems. In its recommended configuration, it uses a so-called "multi-key-v3" encryption mode, in which each disk block is encrypted with one of 64 encryption keys. By default, it encrypts sectors with AES in CBC mode, using an additional AES key to generate IVs.

We configured an encrypted disk with Loop-AES version 3.2b using 128-bit AES encryption in "multi-key-v3" mode. After imaging the contents of RAM, we applied our `keyfind` program, which revealed the 65 AES keys. An attacker could identify which of these keys correspond to which encrypted disk blocks by performing a series of trial decryptions. Then, the attacker could modify the Linux `losetup` utility to mount the encrypted disk with the recovered keys.

Loop-AES attempts to guard against the long-term memory burn-in effects described by Gutmann [23] and others. For each of the 65 AES keys, it maintains two copies of the key schedule in memory, one normal copy and one with each bit inverted. It periodically swaps these copies, ensuring that every memory cell stores a 0 bit for as much time as it stores a 1 bit. Not only does this fail to prevent the memory remanence attacks that we describe here, but it also makes it easier to identify which keys belong to Loop-AES and to recover the keys in the presence of memory errors. After recovering the regular AES key schedules using a program like `keyfind`, the attacker can search the memory image for the inverted key schedules. Since very few programs maintain both regular and inverted key schedules in this way, those keys are highly likely to belong to Loop-AES. Having two related copies of each key schedule provides additional redundancy that can be used to identify which bit positions are likely to contain errors.

## 8 Countermeasures and their Limitations

Memory imaging attacks are difficult to defend against because cryptographic keys that are in active use need to be stored *somewhere*. Our suggested countermeasures focus on discarding or obscuring encryption keys before an adversary might gain physical access, preventing memory-dumping software from being executed on the machine, physically protecting DRAM chips, and possibly making the contents of memory decay more readily.

**Scrubbing memory** Countermeasures begin with efforts to avoid storing keys in memory. Software should overwrite keys when they are no longer needed, and it should attempt to prevent keys from being paged to disk. Runtime libraries and operating systems should clear memory proactively; Chow *et al.* show that this precaution need not be expensive [13]. Of course, these precautions cannot protect keys that must be kept in memory because they are still in use, such as the keys used by encrypted disks or secure web servers.

Systems can also clear memory at boot time. Some PCs can be configured to clear RAM at startup via a destructive Power-On Self-Test (POST) before they attempt to load an operating system. If the attacker cannot bypass the POST, he cannot image the PC's memory with locally-executing software, though he could still physically move the memory chips to different computer with a more permissive BIOS.

**Limiting booting from network or removable media** Many of our attacks involve booting a system via the network or from removable media. Computers can be configured to require an administrative password to boot from these sources. We note, however, that even if a system will boot only from the primary hard drive, an attacker could still swap out this drive, or, in many cases, reset the computer's NVRAM to re-enable booting from removable media.

**Suspending a system safely** Our results show that simply locking the screen of a computer (i.e., keeping the system running but requiring entry of a password before the system will interact with the user) does not protect the contents of memory. Suspending a laptop's state ("sleeping") is also ineffective, even if the machine enters screen-lock on awakening, since an adversary could simply awaken the laptop, power-cycle it, and then extract its memory state. Suspending-to-disk ("hibernating") may also be ineffective unless an externally-held secret is required to resume normal operations.

With most disk encryption systems, users can protect themselves by powering off the machine completely when it is not in use. (BitLocker in "basic" TPM mode remains vulnerable, since the system will automatically mount the disk when the machine is powered on.) Memory contents may be retained for a short period, so the owner should guard the machine for a minute or so after removing power. Though effective, this countermeasure is inconvenient, since the user will have to wait through the lengthy boot process before accessing the machine again.

Suspending can be made safe by requiring a password or other external secret to reawaken the machine, and encrypting the contents of memory under a key derived from the password. The password must be strong (or strengthened), as an attacker who can extract memory contents can then try an offline password-guessing attack. If encrypting all of memory is too expensive, the system could encrypt only those pages or regions containing important keys. Some existing systems can be configured to suspend safely in this sense, although this is often not the default behavior [5].

**Avoiding precomputation** Our attacks show that using precomputation to speed cryptographic operations can make keys more vulnerable. Precomputation tends to lead to redundant storage of key information, which can help an attacker reconstruct keys in the presence of bit errors, as described in Section 5.

Avoiding precomputation may hurt performance, as potentially expensive computations will be repeated. (Disk encryption systems are often implemented on top of OS- and drive-level caches, so they are more performance-sensitive than might be assumed.) Compromises are possible; for example, precomputed values could be cached for a predetermined period of time and discarded if not re-used within that interval. This approach accepts some vulnerability in exchange for reducing computation, a sensible tradeoff in some situations.

**Key expansion** Another defense against key reconstruction is to apply some transform to the key as it is stored in memory in order to make it more difficult to reconstruct in the case of errors. This problem has been considered from a theoretical perspective; Canetti *et al.* [11] define the notion of an *exposure-resilient function* whose input remains secret even if all but some small fraction of the output is revealed, and show that the existence of this primitive is equivalent to the existence of one-way functions.

In practice, suppose we have a key $K$ which is not currently in use but will be needed later. We cannot overwrite the key but we want to make it more resistant to reconstruction. One way to do this is to allocate a large $B$-bit buffer, fill the buffer with random data $R$, then store $K \oplus H(R)$ where $H$ is a hash function such as SHA-256.

Now suppose there is a power-cutting attack which causes $d$ of the bits in this buffer to be flipped. If the hash function is strong, the adversary must search a space of size $\binom{B/2+d}{d}$ to discover which bits were flipped of the roughly $B/2$ that could have decayed. If $B$ is large, this search will be prohibitive even when $d$ is relatively small.

In principle, all keys could be stored in this way, recomputed when in use, and deleted immediately after. Alternatively, we could sometimes keep keys in memory, introducing the precomputation tradeoff discussed above.

For greater protection, the operating system could perform tests to identify memory locations that are especially quick to decay, and use these to store key material.

**Physical defenses** Some of our attacks rely on physical access to DRAM chips or modules. These attacks can be mitigated by physically protecting the memory. For example, DRAM modules could be locked in place inside the machine, or encased in a material such as epoxy to frustrate attempts to remove or access them. Similarly, the system could respond to low temperatures or opening of the computer's case by attempting to overwrite memory, although these defenses require active sensor systems with their own backup power supply. Many of these techniques are associated with specialized tamper-resistant hardware such as the IBM 4758 coprocessor [18, 41] and could add considerable cost to a PC. However, a small amount of memory soldered to a motherboard could be added at relatively low cost.

**Architectural changes** Some countermeasures try to change the machine's architecture. This will not help on existing machines, but it might make future machines more secure.

One approach is to find or design DRAM systems that lose their state quickly. This might be difficult, given the tension between the desire to make memory decay quickly and the desire to keep the probability of decay within a DRAM refresh interval vanishingly small.

Another approach is to add key-store hardware that erases its state on power-up, reset, and shutdown. This would provide a safe place to put a few keys, though precomputation of derived keys would still pose a risk.

Others have proposed architectures that would routinely encrypt the contents of memory for security purposes [28, 27, 17]. These would apparently prevent the attacks we describe, as long as the encryption keys were destroyed on reset or power loss.

**Encrypting in the disk controller** Another approach is to encrypt data in the hard disk controller hardware, as in Full Disk Encryption (FDE) systems such as Seagate's "DriveTrust" technology [38].

In its basic form, this approach uses a write-only *key register* in the disk controller, into which the software can write a symmetric encryption key. Data blocks are encrypted, using the key from the key register, before writing to the disk. Similarly, blocks are decrypted after reading. This allows encrypted storage of all blocks on a disk, without any software modifications beyond what is required to initialize the key register.

This approach differs from typical disk encryption systems in that encryption and decryption are done by the disk controller rather than by software in the main CPU, and that the main encryption keys are stored in the disk controller rather than in DRAM.

To be secure, such a system must ensure that the key register is erased whenever a new operating system is booted on the computer; otherwise, an attacker can reboot into a malicious kernel that simply reads the disk contents. For similar reasons, the system must also ensure that the key register is erased whenever an attacker attempts to move the disk controller to another computer (even if the attacker maintains power while doing so).

Some systems built more sophisticated APIs, implemented by software on the disk controller, on top of this basic facility. Such APIs, and their implementation, would require further security analyses.

We have not evaluated any specific systems of this type. We leave such analyses for future work.

**Trusted computing** Trusted Computing hardware, in the form of Trusted Platform Modules (TPMs) [42] is now deployed in some personal computers. Though useful against some attacks, today's Trusted Computing hardware does not seem to prevent the attacks described here.

Deployed TCG TPMs do not implement bulk encryption. Instead, they monitor boot history in order to decide (or help other machines decide) whether it is safe to store a key in RAM. If a software module wants to use a key, it can arrange that the usable form of that key will not be stored in RAM unless the boot process has gone as expected [31]. However, once the key is stored in RAM, it is subject to our attacks. TPMs can prevent a key from being loaded into memory for use, but they cannot prevent it from being captured once it is in memory.

## 9 Conclusions

Contrary to popular belief, DRAMs hold their values for surprisingly long intervals without power or refresh. Our experiments show that this fact enables a variety of security attacks that can extract sensitive information such as cryptographic keys from memory, despite the operating system's efforts to protect memory contents. The attacks we describe are practical—for example, we have used them to defeat several popular disk encryption systems.

Other types of software may be similarly vulnerable. DRM systems often rely on symmetric keys stored in memory, which may be recoverable using the techniques outlined in our paper. As we have shown, SSL-enabled web servers are vulnerable, since they often keep in memory private keys needed to establish SSL sessions. Furthermore, methods similar to our key-finder would likely be effective for locating passwords, account numbers, or other sensitive data in memory.

There seems to be no easy remedy for these vulnerabilities. Simple software changes have benefits and drawbacks; hardware changes are possible but will require time and expense; and today's Trusted Computing technologies cannot protect keys that are already in memory. The risk seems highest for laptops, which are often taken out in public in states that are vulnerable to our attacks. These risks imply that disk encryption on laptops, while beneficial, does not guarantee protection.

Ultimately, it might become necessary to treat DRAM as untrusted, and to avoid storing sensitive data there, but this will not be feasible until architectures are changed to give software a safe place to keep its keys.

## Acknowledgments

We thank Andrew Appel, Jesse Burns, Grey David, Laura Felten, Christian Fromme, Dan Good, Peter Gutmann, Benjamin Mako Hill, David Hulton, Brie Ilenda, Scott Karlin, David Molnar, Tim Newsham, Chris Palmer, Audrey Penven, David Robinson, Kragen Sitaker, N.J.A. Sloane, Gregory Sutter, Sam Taylor, Ralf-Philipp Weinmann, and Bill Zeller for their helpful comments, suggestions, and contributions. Without them, this paper would not have been possible.

This material is based in part upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Calandrino performed this research under an appointment to the Department of Homeland Security (DHS) Scholarship and Fellowship Program, administered by the Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy (DOE) and DHS. ORISE is managed by Oak Ridge Associated Universities (ORAU) under DOE contract number DE-AC05-06OR23100.

All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DHS, DOE, or ORAU/ORISE.

## References

[1] Advanced Encryption Standard. National Institute of Standards and Technology, FIPS-197, Nov. 2001.

[2] ANDERSON, R. *Security Engineering: A Guide to Building Dependable Distributed Systems*, first ed. Wiley, Jan. 2001, p. 282.

[3] ANVIN, H. P. SYSLINUX. http://syslinux.zytor.com/.

[4] ARBAUGH, W., FARBER, D., AND SMITH, J. A secure and reliable bootstrap architecture. In *Proc. IEEE Symp. on Security and Privacy* (May 1997), pp. 65–71.

[5] BAR-LEV, A. Linux, Loop-AES and optional smartcard based disk encryption. http://wiki.tuxonice.net/EncryptedSwapAndRoot, Nov. 2007.

[6] BARRY, P., AND HARTNETT, G. *Designing Embedded Networking Applications: Essential Insights for Developers of Intel IXP4XX Network Processor Systems*, first ed. Intel Press, May 2005, p. 47.

[7] BLÖMER, J., AND MAY, A. New partial key exposure attacks on RSA. In *Proc. CRYPTO 2003* (2003), pp. 27–43.

[8] BOILEAU, A. Hit by a bus: Physical access attacks with Firewire. Presentation, Ruxcon, 2006.

[9] BONEH, D., DURFEE, G., AND FRANKEL, Y. An attack on RSA given a small fraction of the private key bits. In *Advances in Cryptology – ASIACRYPT '98* (1998), pp. 25–34.

[10] CALIFORNIA STATUTES. Cal. Civ. Code §1798.82, created by S.B. 1386, Aug. 2002.

[11] CANETTI, R., DODIS, Y., HALEVI, S., KUSHILEVITZ, E., AND SAHAI, A. Exposure-resilient functions and all-or-nothing transforms. In *Advances in Cryptology – EUROCRYPT 2000* (2000), vol. 1807/2000, pp. 453–469.

[12] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1* (Dec. 2003), 50–60.

[13] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium* (Aug. 2005), pp. 331–346.

[14] COPPERSMITH, D. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology 10*, 4 (1997), 233–260.

[15] DORNSEIF, M. 0wned by an iPod. Presentation, PacSec, 2004.

[16] DORNSEIF, M. Firewire – all your memory are belong to us. Presentation, CanSecWest/core05, May 2005.

[17] DWOSKIN, J., AND LEE, R. B. Hardware-rooted trust for secure key management and transient trust. In *Proc. 14th ACM Conference on Computer and Communications Security* (Oct. 2007), pp. 389–400.

[18] DYER, J. G., LINDEMANN, M., PEREZ, R., SAILER, R., VAN DOORN, L., SMITH, S. W., AND WEINGART, S. Building the IBM 4758 secure coprocessor. *Computer 34* (Oct. 2001), 57–66.

[19] ECKSTEIN, K., AND DORNSEIF, M. On the meaning of 'physical access' to a computing device: A vulnerability classification of mobile computing devices. Presentation, NATO C3A Workshop on Network-Enabled Warfare, Apr. 2005.

[20] ELSON, J., AND GIROD, L. Fusd – a Linux framework for user-space devices. http://www.circlemud.org/ jelson/software/fusd/.

[21] FERGUSON, N. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. http://www.microsoft.com/downloads/details.aspx?FamilyID=131dae03-39ae-48be-a8d6-8b0034c92555, Aug. 2006.

[22] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *Proc. 6th USENIX Security Symposium* (July 1996), pp. 77–90.

[23] GUTMANN, P. Data remanence in semiconductor devices. In *Proc. 10th USENIX Security Symposium* (Aug. 2001), pp. 39–54.

[24] IEEE 1619 SECURITY IN STORAGE WORKING GROUP. IEEE P1619/D19: Draft standard for cryptographic protection of data on block-oriented storage devices, July 2007.

[25] INTEL CORPORATION. Preboot Execution Enviroment (PXE) specification version 2.1, Sept. 1999.

[26] KENT, C. Draft proposal for tweakable narrow-block encryption. https://siswg.net/docs/LRW-AES-10-19-2004.pdf, 2004.

[27] LEE, R. B., KWAN, P. C., MCGREGOR, J. P., DWOSKIN, J., AND WANG, Z. Architecture for protecting critical secrets in microprocessors. In *Proc. Intl. Symposium on Computer Architecture* (2005), pp. 2–13.

[28] LIE, D., THEKKATH, C. A., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural support for copy and tamper resistant software. In *Symp. on Architectural Support for Programming Languages and Operating Systems* (2000), pp. 168–177.

[29] LINK, W., AND MAY, H. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv für Elektronik und Übertragungstechnik 33* (June 1979), 229–235.

[30] LISKOV, M., RIVEST, R. L., AND WAGNER, D. Tweakable block ciphers. In *Advances in Cryptology – CRYPTO 2002* (2002), pp. 31–46.

[31] MACIVER, D. Penetration testing Windows Vista BitLocker drive encryption. Presentation, Hack In The Box, Sept. 2006.

[32] NATIONAL CONFERENCE OF STATE LEGISLATURES. State security breach notification laws. http://www.ncsl.org/programs/lis/cip/priv/breachlaws.htm, Jan. 2008.

[33] PETTERSSON, T. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp, Aug. 2007.

[34] PTACEK, T. Recover a private key from process memory. http://www.matasano.com/log/178/recover-a-private-key-from-process-memory/.

[35] ROGAWAY, P. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology – ASIACRYPT 2004* (2004), pp. 16–31.

[36] RSA LABORATORIES. PKCS #1 v2.1: RSA cryptography standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[37] SCHEICK, L. Z., GUERTIN, S. M., AND SWIFT, G. M. Analysis of radiation effects on individual DRAM cells. *IEEE Transactions on Nuclear Science 47* (Dec. 2000), 2534–2538.

[38] SEAGATE CORPORATION. Drivetrust technology: A technical overview. http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf.

[39] SHAMIR, A., AND VAN SOMEREN, N. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science 1648* (1999), 118–124.

[40] SKOROBOGATOV, S. Low-temperature data remanence in static RAM. University of Cambridge Computer Laborary Technical Report No. 536, June 2002.

[41] SMITH, S. W. *Trusted Computing Platforms: Design and Applications*, first ed. Springer, 2005.

[42] TRUSTED COMPUTING GROUP. Trusted Platform Module specification version 1.2, July 2007.

[43] VIDAS, T. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice 1* (Dec. 2006), 315–323.

[44] WEINMANN, R.-P., AND APPELBAUM, J. Unlocking FileVault. Presentation, 23rd Chaos Communication Congress, Dec. 2006.

[45] WEINMANN, R.-P., AND APPELBAUM, J. VileFault. http://vilefault.googlecode.com/, Jan. 2008.

[46] WYNS, P., AND ANDERSON, R. L. Low-temperature operation of silicon dynamic random-access memories. *IEEE Transactions on Electron Devices 36* (Aug. 1989), 1423–1428.

# The Practical Subtleties of Biometric Key Generation

Lucas Ballard
*Department of Computer Science*
*The Johns Hopkins University*

Seny Kamara
*Department of Computer Science*
*The Johns Hopkins University*

Michael K. Reiter
*Department of Computer Science*
*University of North Carolina at Chapel Hill*

## Abstract

The inability of humans to generate and remember strong secrets makes it difficult for people to manage cryptographic keys. To address this problem, numerous proposals have been suggested to enable a human to repeatably generate a cryptographic key from her biometrics, where the strength of the key rests on the assumption that the measured biometrics have high entropy across the population. In this paper we show that, despite the fact that several researchers have examined the security of BKGs, the common techniques used to argue the security of practical systems are lacking. To address this issue we reexamine two well known, yet sometimes misunderstood, security requirements. We also present another that we believe has not received adequate attention in the literature, but is essential for practical biometric key generators. To demonstrate that each requirement has significant importance, we analyze three published schemes, and point out deficiencies in each. For example, in one case we show that failing to meet a requirement results in a construction where an attacker has a 22% chance of finding ostensibly 43-bit keys on her *first* guess. In another we show how an attacker who compromises a user's cryptographic key can then infer that user's biometric, thus revealing any other key generated using that biometric. We hope that by examining the pitfalls that occur continuously in the literature, we enable researchers and practitioners to more accurately analyze proposed constructions.

## 1 Introduction

While cryptographic applications vary widely in terms of assumptions, constructions, and goals, all require cryptographic keys. In cases where a computer should not be trusted to protect cryptographic keys—as in laptop file encryption, where keeping the key on the laptop obviates the utility of the file encryption—the key must be input by its human operator. It is well known, however, that humans have difficulty choosing and remembering strong secrets (e.g., [2, 14]). As a result, researchers have devoted significant effort to finding input that has sufficient unpredictability to be used in cryptographic applications, but that remains easy for humans to regenerate reliably. One of the more promising suggestions in this direction are biometrics—characteristics of human physiology or behavior. Biometrics are attractive as a means for key generation as they are easily reproducible by the legitimate user, yet potentially difficult for an adversary to guess.

There have been numerous proposals for generating cryptographic keys from biometrics (e.g., [33, 34, 28]). At a high level, these Biometric Cryptographic Key Generators, or BKGs, follow a similar design: during an enrollment phase, biometric samples from a user are collected; statistical functions, or *features*, are applied to the samples; and some representation of the output of these features is stored in a data structure called a biometric *template*. Later, the same user can present another sample, which is processed with the stored template to reproduce a key. A different user, however, should be unable to produce that key. Since the template itself is generally stored where the key is used (e.g., in a laptop file encryption application, on the laptop), a template must not leak any information about the key that it is used to reconstruct. That is, the threat model admits the capture of the template by the adversary; otherwise the template could be the cryptographic key itself, and biometrics would not be needed to reconstruct the key at all.

Generally, one measures the strength of a cryptographic key by its *entropy*, which quantifies the amount of uncertainty in the key from an adversary's point of view. If one regards a key generator as drawing an element uniformly at random from a large set, then the entropy of the keys can be easily computed as the base-two logarithm of the size of the set. Computing the entropy of keys output by a concrete instantiation of a key generator, however, is non-trivial because "choosing uniformly at random" is difficult to achieve in practice. This is in part due to the fact that the key generator's source of randomness may be based on information that is leaked by external sources. For instance, an oft-cited flaw in Kerberos version 4 allowed adversaries to guess ostensibly 56-bit DES session keys in only $2^{20}$ guesses [13]. The problem stemmed from the fact that the seeding information input to the key generator was related to information that could be easily inferred by an adversary. In other words, this *auxiliary information* greatly reduced the entropy of the key space.

In the case of biometric key generators, where the randomness used to generate the keys comes from a user's biometric and is a function of the particular features used by the system, the aforementioned problems are compounded by several factors. For instance, in the case of certain biometric modalities, it is known that population statistics can be strong indicators of a specific user's biometric [10, 36, 3]. In other words, depending on the type of biometric and the set of features used by the BKG, access to population statistics can greatly reduce the entropy of a user's biometric, and consequently, reduce the entropy of her key. Moreover, templates could also leak information about the key. To complicate matters, in the context of biometric key generation, in addition to evaluating the strength of the key, one must also consider the privacy implications associated with using biometrics. Indeed, the protection of a user's biometric information is crucial, not only to preserve privacy, but also to enable that user to reuse the biometric key generator to manage a new key. We argue that this concern for privacy mandates not only that the template protect the biometric, but also that the keys output by a BKG not leak information about the biometric. Otherwise, the compromise of a key might render the user's biometric unusable for key generation thereafter.

The goal of this work is to distill the seemingly intertwined and complex security requirements of biometric key generators into a small set of requirements that facilitate *practical* security analyses by designers. Specifically, the contributions of this paper are:

I. The specification of three practical requirements that allow designers to ensure that a BKG ensures the privacy of a user's biometric and generates keys that are suitable for cryptographic applications.

II. The analyses of three published BKGs. These are contributions in their own right, but more importantly serve as concrete evidence of the importance of the requirements.

III. The description of *Guessing Distance*, a new heuristic measure that, given empirical data, can quickly estimate the security afforded by a BKG.

IV. Discussion of common pitfalls and subleties in current standards for empirical evaluation.

Throughout this paper we focus on the importance of considering adversaries who have access to public information, such as templates, when performing security evaluations. We hope that our observations will promote critical analyses of BKGs and temper the spread of flawed (or incorrectly evaluated) proposals.

## 2 Related Work

To our knowledge, Soutar and Tomko [34] were the first to propose biometric key generation. Davida et al. [9] proposed an approach that uses iris codes, which are believed to have the highest entropy of all commonly-used biometrics. However, iris code collection can be considered somewhat invasive and the use of majority-decoding for error correction—a central ingredient of the Davida et al. approach—has been argued to have limited use in practice [16].

Monrose et al. proposed the first practical BKG that exploits behavioral (versus physiological) biometrics for key generation [29]. Their technique uses keystroke latencies to increase the entropy of standard passwords. Their construction yields a key at least as secure as the password alone, and an empirical analysis showed that their approach increases the workload of an attacker by a multiplicative factor of up to $2^{15}$. A similar approach was used to generate cryptographic keys from voice [28, 27]. Many constructions followed those of Monrose et al., using biometrics such as face [15], fingerprints [33, 39], handwriting [40, 17] and iris codes [16, 45]. Unfortunately, many are susceptible to attacks. Hill-climbing attacks have been leveraged against fingerprint, face, and handwriting-based biometric systems [1, 37, 43] by exploiting information leaked during the reconstruction of the key from the biometric template.

There has also been an emergence of generative attacks against biometrics [5, 23], which use auxiliary information such as population statistics along with limited information about a target user's biometric. The attacks we present in this paper are different from generative attacks because we assume that adversaries only have access to templates and auxiliary information. Our attacks, therefore, capture much more limited, and arguably more realistic, adversaries. Despite such limited information, we show how an attacker can recover a target user's key with high likelihood.

There has also been recent theoretical work to formalize particular aspects of biometric key generators. The idea of fuzzy cryptography was first introduced by Juels and Wattenberg [21], who describe a commitment scheme that supports noise-tolerant decommitments. In

Section 7 we provide a concrete analysis of a published construction that highlights the pitfalls of using fuzzy commitments as biometric key generators. Further work included a fuzzy vault [20], which was later analyzed as an instance of a secure-sketch that can be used to build fuzzy extractors [11, 6, 12, 22]. Fuzzy extractors treat biometric features as non-uniformly distributed, error-prone sources and apply error-correction algorithms and randomness extractors [18, 30] to generate random strings.

Fuzzy cryptography has made important contributions by specifying formal security definitions with which BKGs can be analyzed. Nevertheless, there remains a gap between theoretical soundness and practical systems. For instance, while fuzzy extractors can be effectively used as a component in a larger biometric key generation system, they do not capture all the practical requirements of a BKG. In particular, it is unclear whether known constructions can correct the kinds of errors typically generated by humans, especially in the case of behavioral biometrics. Moreover, fuzzy extractors require biometric inputs with high min-entropy but do not address how to select features that achieve this requisite level of entropy. Since this is an inherently empirical question, much of our work is concerned with how to *experimentally* evaluate the entropy available in a biometric.

Lastly, Jain et al. enumerate possible attacks against biometric templates and discuss several practical approaches that increase template security [19]. Similarly, Mansfield and Wayman discuss a set of best practices that may be used to measure the security and usability of biometric systems [24]. While these works describe specific attacks and defenses against systems, they do not address biometric key generators and the unique requirements they demand.

## 3 Biometric Key Generators

Before we can argue about how to accurately assess biometric key generators (BKGs), we first define the algorithms and components associated with a BKG. These definitions are general enough to encompass most proposed BKGs.

BKGs are generally composed of two algorithms, an enrollment algorithm (Enroll) and a key-generation algorithm (KeyGen):

- $\mathsf{Enroll}(\mathcal{B}_1,\ldots,\mathcal{B}_\ell)$: The enroll algorithm is a probabilistic algorithm that accepts as input a number of biometric samples ($\mathcal{B}_1,\ldots,\mathcal{B}_\ell$), and outputs a template ($\mathcal{T}$) and a cryptographic key ($\mathcal{K}$). In the event that $\mathcal{B}_1,\ldots,\mathcal{B}_\ell$ do not meet some predetermined criteria, the enroll algorithm might output the failure symbol $\perp$.
- $\mathsf{KeyGen}(\mathcal{B},\mathcal{T})$: The key generation algorithm accepts as input one biometric sample ($\mathcal{B}$), and a template ($\mathcal{T}$). The algorithm outputs either a cryptographic key ($\mathcal{K}$), or the failure symbol $\perp$ if $\mathcal{B}$ cannot be used to create a key.

The enrollment algorithm estimates the variation inherent to a particular user's biometric reading and computes information needed to error-correct a new sample that is sufficiently close to the enrollment samples. Enroll encodes this information into a template and outputs the template and the associated key. The key-generation algorithm uses the template output by the enrollment algorithm and a new biometric sample to output a key. If the provided sample is sufficiently similar to those provided during enrollment, then KeyGen and Enroll output the same keys.

Generally speaking, there are four classes of information associated with a BKG.

- The Biometric ($\mathcal{B}$): A biometric is a measurement of a person's behavior or physiology. A BKG extracts $\mathcal{B}$ as algorithmically interpretable representations (e.g., a set of signals). The BKG typically applies statistical functions, or features ($\phi_1,\ldots,\phi_n$), to the representations, and uses the output to either derive [17, 41] or lock [33, 16, 38] a cryptographic key.
- A Template ($\mathcal{T}$): A template is any piece of information that is stored on the system for the purpose of re-generating the cryptographic key. Templates are generally created during an enrollment process and stored so that a user can easily recreate her key. For all practical purposes, templates must be considered publicly available. Note that this assumption implies that more standard biometric templates, which are typically employed for authentication purposes and are simply the encoding of a biometric [42], cannot be used securely in this setting.
- The Key ($\mathcal{K}$): A cryptographic key that is derived from (or locked by) one or more biometric samples during an enrollment phase. The key may later be regenerated using another biometric sample that is "close" to the original samples and the template that was also output during enrollment.
- Auxiliary Information ($\mathcal{A}$): Auxiliary information encompasses any public information not intended to be used for key-derivation purposes but that is still readily available to an adversary. Auxiliary information is specified with respect to one user and includes any biometric, template, or key other than those associated with the user in question. It could

also include any other information about the environment that might leak information about the biometric, or results of using the key.

For the remainder of the paper if a component of a BKG is associated with a specific user, then we subscript the information with the user's unique identifier. So, for example, $\mathcal{B}_u$, is $u$'s biometric and $\mathcal{A}_{\bar{u}}$ is auxiliary information derived from all users $u' \neq u$.

## 3.1 Evaluation Recommendations

At a high level, the evaluation of a BKG requires designers to show that two properties hold: *correctness* and *security*. Intuitively, a scheme that achieves correctness is one that is usable for a high percentage of the population. That is, the biometric of choice can be reliably extracted to within some threshold of tolerance, and when combined with the template the correct key is output with high probability. As correctness is well understood, and is always presented when discussing the feasibility of a proposed BKG, we do not address it further.

In the context of biometric key generation, security is not as easily defined as correctness. Loosely speaking, a secure BKG outputs a key that "looks random" to any adversary that cannot guess the biometric. In addition, the templates and keys derived by the BKG should not leak any information about the biometric that was used to create them. We enumerate a set of three security requirements for biometric key generators, and examine the components that should be analyzed mathematically (i.e., the template and key) and empirically (i.e., the biometric and auxiliary information). While the necessity of the first two requirements has been understood to some degree, we will highlight and analyze how previous evaluations of these requirements are lacking. Additionally, we discuss a requirement that is often overlooked in the practical literature, but one which we believe is necessary for a secure and practical BKG.

We consider a BKG secure if it meets the following three requirements for each enrollable user in a population:

- *Key Randomness* (REQ-KR): The keys output by a BKG appear random to any adversary who has access to auxiliary information and the template used to derive the key. For instance, we might require that the key be statistically or computationally indistinguishable from random.

- *Weak Biometric Privacy* (REQ-WBP): An adversary learns no useful information about a biometric given auxiliary information and the template used to derive the key. For instance, no computationally bounded adversary should be able to compute any function of the biometric.

- *Strong Biometric Privacy* (REQ-SBP): An adversary learns no useful information about a biometric given auxiliary information, the template used to derive the key, and the key itself. For instance, no computationally bounded adversary should be able to compute any function of the biometric.

The necessity of REQ-KR and REQ-WBP is well known, and indeed many proposals make some sort of effort to argue security along these lines (see, e.g., [29, 11]). However, many different approaches are used to make these arguments. Some take a cryptographically formal approach, whereas others provide an empirical evaluation aimed at demonstrating that the biometrics and the generated keys have high entropy. Unfortunately, the level of rigor can vary between works, and differences in the ways REQ-KR and REQ-WBP are typically argued make it difficult to compare approaches. Also, it is not always clear that the empirical assumptions required by the cryptographic algorithms of the BKG can be met in practice.

Even more problematic is that many approaches for demonstrating biometric security merely provide some sort of measure of entropy of a biometric (or key) based on variation across a population. For example, one common approach is to compute biometric features for each user in a population, and compute the entropy over the output of these features. However, such analyses are generally lacking on two counts. For one, if the correlation between features is not accounted for, the reported entropy of the scheme being evaluated could be much higher than what an adversary must overcome in practice. Second, such techniques fail to compute entropy as a function of the biometric templates, which we argue should be assumed to be publicly available. Consequently, such calculations would declare a BKG "secure" even if, say, the template leaked information about the derived key. For example, suppose that a BKG uses only one feature and simply quantizes the feature space, outputting as a key the region of the feature space that contains the majority of the measurements of a specific user's feature. The quantization is likely to vary between users, and so the partitioning information would need to be stored in each user's template. Possession of the template thus reduces the set of possible keys, as it defines how the feature space is partitioned.

As far as we know, the notion of Strong Biometric Privacy (REQ-SBP) has only been considered recently, and only in a theoretical setting [12]. Even the original definitions of fuzzy extractors [11, Definition 3] do not explicitly address this requirement. Unfortunately, REQ-SBP has also largely been ignored by the designers of practical systems. Perhaps this oversight is due to lack of perceived practical motivation—it is not immediately clear that a key could be used to reveal a user's biometric. Indeed, to our knowledge, there have been few, if any, concrete attacks that have used keys and templates to infer a user's biometric. We observe, however, that it is precisely practical situations that motivate such a requirement; keys output by a BKG could be revealed for any number of reasons in real systems (e.g., side-channel attacks against encryption keys, or the verification of a MAC). If a key can be used to derive the biometric that was used to generate the key, then key recovery poses a severe privacy concern. Moreover, key compromise would then preclude a user from using this biometric in a BKG ever again, as the adversary would be able to recreate any key the user makes thereafter. Therefore, in Section 7 we provide specific practical motivation for this requirement by describing an attack against a well-accepted BKG. The attack combines the key and a template to infer a user's biometric.

In what follows, we provide practical motivation for the importance of each of our three requirements by analyzing three published BKGs. It is not our goal to fault specific constructions, but instead to critique evaluation techniques that have become standard practice in the field. We chose to analyze these specific BKGs because each was argued to be secure using "standard" techniques. However, we show that since these techniques do not address important requirements, each of these constructions exhibit significant weaknesses despite security arguments to the contrary.

## 4 Biometrics and "Entropy"

Before continuing further, we note that analyzing the security of a biometric key generator is a challenging task. A comprehensive approach to biometric security should consider sources of auxiliary information, as well as the impact of human forgers. Though it may seem impractical to consider the latter as a potential threat to a standard key generator, skilled humans can be used to generate initial forgeries that an algorithmic approach can then leverage to undermine the security of the BKG.

To this point, research has accepted this "adversarial multiplicity" without examining the consequences in great detail. Many works (e.g., [33, 29, 17, 15, 40, 16]) report both False Accept Rates (i.e., how often a human can forge a biometric) and an estimate of key entropy (i.e., the supposed difficulty an algorithm must overcome in order to guess a key) without specifically identifying the intended adversary. In this work, we focus on algorithmic adversaries given their importance in offline guessing attacks, and because we have already addressed the importance of considering human-aided forgeries [4, 5]. While our previous work did not address biometric key generators specifically, those lessons apply equally to this case.

The security of biometric key generators in the face of algorithmic adversaries has been argued in several different ways, and each approach has advantages and disadvantages. Theoretical approaches (e.g., [11, 6]) begin by assuming that the biometrics have high adversarial min-entropy (i.e., conditioned on all the auxiliary information available to an adversary, the entropy of the biometric is still high) and then proceed to distill this entropy into a key that is statistically close to uniform. However, in practice, it is not always clear how to estimate the uncertainty of a biometric. In more practical settings, guessing entropy [25] has been used to measure the strength of keys (e.g., [29, 27, 10]), as it is easily computed from empirical data. Unfortunately, as we demonstrate shortly, guessing entropy is a summary statistic and can thus yield misleading results when computed over skewed distributions. Yet another common approach (e.g., [31, 7, 16, 41, 17]), which has lead to somewhat misleading views on security, is to argue key strength by computing the Shannon entropy of the key distribution over a population. More precisely, if we consider a BKG that assigns the key $\mathcal{K}_u$ to a user $u$ in a population $P$, then it is considered "secure" if the entropy of the distribution $\mathcal{P}(\mathcal{K}) = |\{u \in P : \mathcal{K}_u = \mathcal{K}\}|/|P|$ is high. We note, however, that the entropy of the previous distribution measures only key uniqueness and says nothing about how difficult it is for an adversary to guess the key. In fact, it is not difficult to design BKGs that output keys with maximum entropy in the previous sense, but whose keys are easy for an adversary to guess; setting $\mathcal{K}_u = u$ is a trivial example.

To address these issues, we present a new measure that is easy to compute empirically and that estimates the difficulty an adversary will have in guessing the output of a distribution given some related auxiliary distribution. It can be used to empirically estimate the entropy of a biometric for any adversary that assumes the biometric is distributed similarly to the auxiliary distribution. Our proposition, *Guessing Distance*, involves determining the number of guesses that an adversary must make to identify a biometric or key, and how the number of guesses are reduced in light of various forms of auxiliary information.

### 4.1 Guessing Distance

We assume that a specific user $u$ induces a distribution $\mathcal{U}$ over a finite, $n$-element set $\Omega$. We also assume that an adversary has access to population statistics that also induce a distribution, $\mathcal{P}$, over $\Omega$. $\mathcal{P}$ could be computed from the distributions of other users $u' \neq u$. We seek to quantify how useful $\mathcal{P}$ is at predicting $\mathcal{U}$. The specification of $\Omega$ varies depending on the BKG being analyzed; $\Omega$ could be a set of biometrics, a set of possible feature outputs, or a set of keys. It is up to system designers to use the specification of $\Omega$ that would most likely be used by an adversary. For instance, if the output of features are easier to guess than a biometric, then $\Omega$ should be defined as the set of possible feature outputs. Although at this point we keep the definition of $\mathcal{P}$ and $\mathcal{U}$ abstract, it is important when assessing the security of a construction to take as much auxiliary information as possible into account when estimating $\mathcal{P}$. We return in Section 5 with an example of such an analysis.

We desire a measure that estimates the number of *guesses* that an adversary will make to find the high-probability elements of $\mathcal{U}$, when guessing by enumerating $\Omega$ starting with the most likely elements as prescribed by $\mathcal{P}$. That is, our measure need not precisely capture the distance between $\mathcal{U}$ and $\mathcal{P}$ (as might, say, $L_1$-distance or Relative Entropy), but rather must capture simply $\mathcal{P}$'s ability to predict the most likely elements as described by $\mathcal{U}$ [1]. Given a user's distribution $\mathcal{U}$, and two (potentially different) population distributions $\mathcal{P}_1$ and $\mathcal{P}_2$, we would like the distance between $\mathcal{U}$ and $\mathcal{P}_1$ and $\mathcal{U}$ and $\mathcal{P}_2$ to be the same if and only if $\mathcal{P}_1$ and $\mathcal{P}_2$ prescribe the same guessing strategy for a random variable distributed according to $\mathcal{U}$. For example, consider the distributions $\mathcal{U}$, $\mathcal{P}_1$ and $\mathcal{P}_2$, and the element $\omega \in \Omega$ such that $\mathcal{P}_1(\omega) = .9$, $\mathcal{P}_2(\omega) = .8$, and $\mathcal{U}(\omega) = 1$. Here, an adversary with access to $\mathcal{P}_1$ would require the same number of guesses to find $\omega$ as an adversary with access to $\mathcal{P}_2$ (one). Thus, we would like the distance between $\mathcal{U}$ and $\mathcal{P}_1$ and between $\mathcal{U}$ and $\mathcal{P}_2$ to be the same.

**Guessing Distance.** Let $\omega^* = \mathrm{argmax}_{\omega\in\Omega}\mathcal{U}(\omega)$. Let $L_{\mathcal{P}} = (\omega_1, \ldots, \omega_n)$ be the elements of $\Omega$ ordered such that $\mathcal{P}(\omega_i) \geq \mathcal{P}(\omega_{i+1})$ for all $i \in [1, n-1]$. Define $t^-$ and $t^+$ to be the smallest index and largest index $i$ such that $|\mathcal{P}(\omega_i) - \mathcal{P}(\omega^*)| \leq \delta$. The Guessing Distance between $\mathcal{U}$ and $\mathcal{P}$ with tolerance $\delta$ is defined as:

$$\mathsf{GD}_\delta(\mathcal{U}, \mathcal{P}) = \log \frac{t^- + t^+}{2}$$

Guessing Distance measures the number of guesses that an adversary who assumes that $\mathcal{U} \approx \mathcal{P}$ makes before guessing the most likely element as prescribed by $\mathcal{U}$ (that is, $\omega^*$)[2]. We take the average over $t^-$ and $t^+$ as it may be the case that several elements may have similar probability masses under $\mathcal{P}$. In such a situation, the ordering of $L_{\mathcal{P}}$ may be ambiguous, so we report an average measure across all equivalent orderings. As $\mathcal{U}$ and $\mathcal{P}$ will typically be empirical estimates, we use a tolerance $\delta$ to offset small measurement errors when grouping elements of similar probability masses. The subscript $\delta$ is ignored if $\delta = 0$.

**Discussion.** Intuitively, one can see that this definition makes sense by considering the following three cases: (1) $\mathcal{P}$ is a good indicator of $\mathcal{U}$ (i.e., $\omega^* = \omega_1$); (2) $\mathcal{P}$ is uniform; and (3) $\mathcal{P}$ is a poor indicator of $\mathcal{U}$ (i.e., $\omega^* = \omega_n$). In case (1) the adversary clearly benefits from using $\mathcal{P}$ to guess $\omega^*$, and this relation is captured as $\mathsf{GD}(\mathcal{U}, \mathcal{P}) = \log 1 = 0$. In case (2), the adversary learns no information about $\mathcal{U}$ from $\mathcal{P}$ and thus would be expected to search half of $\Omega$ before guessing the correct value; indeed $\mathsf{GD}(\mathcal{U}, \mathcal{P}) = \log \frac{1+n}{2}$. Finally, in case (3), a search algorithm based on $\mathcal{P}$ would need to enumerate all of $\Omega$ before finding $\omega^*$, and this is reflected by $\mathsf{GD}(\mathcal{U}, \mathcal{P}) = \log \frac{n+n}{2} = \log |\Omega|$.

An important characteristic of GD is that it compares two probability distributions. This allows for a more fine-tuned evaluation as one can compute GD for each user in the population. To see the overall strength of a proposed approach, one might report a CDF of the GD's for each user, or report the minimum over all GD's in the population.

Guessing Distance is superficially similar to Guessing Entropy [25], which is commonly used to compute the expected number of guesses it takes to find an average element in a set assuming an optimal guessing strategy (i.e., first guessing the element with the highest likelihood, followed by guessing the element with the second highest likelihood, etc.) Indeed, one might view Guessing Distance as an extension of Guessing Entropy (see Appendix A); however, we prefer Guessing Distance as a measure of security as it provides more information about non-uniform distributions over a key space. For such distributions, Guessing Entropy is increased by the elements that have a low probability, and thus might not provide as conservative an estimate of security as desired. Guessing Distance, on the other hand, can be computed for each user, which brings to light the insecurity afforded by a non-uniform distribution. We provide a concrete example of such a case in Appendix A.

## 5 The Impact of Public Information on Key Randomness

We now show why templates play a crucial role in the computation of key entropy (REQ-KR from Section 3). Our analysis brings to light two points: first that templates, and in particular, error-correction information, can indeed leak a substantial amount of information about a key, and thus must be considered when computing key entropy. Second, we show how standard approaches to computing key entropy, even if they were to take templates into account, must be conducted with care to avoid common pitfalls. Through our analysis we demonstrate the flexibility and utility of Guessing Distance. While we focus here on a specific proposal by Vielhauer and Steinmetz [40, 41], we argue that our results are generally applicable to a host of similar proposals (see, e.g., [44, 7, 35, 17]) that use per-user feature-space quantization for error correction. This complicates the calculation of entropy and brings to light common pitfalls.

The construction works as follows. Given 50 features $\phi_1, \ldots, \phi_{50}$ [40] that map biometric samples to the set of non-negative integers, and $\ell$ enrollment samples $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$, let $\Delta_i$ be the difference between the minimum and maximum value of $\phi_i(\mathcal{B}_1), \ldots, \phi_i(\mathcal{B}_\ell)$, expanded by a small tolerance. The scheme partitions the output range of $\phi_i$ into $\Delta_i$-length segments. The key is derived by letting $L_i$ be the smallest integer in the segment that contains the user's samples, computing $\Gamma_i = L_i \bmod \Delta_i$, and setting the $i^{\text{th}}$ key element $c_i = \left\lfloor \frac{\phi_i(\mathcal{B}_1) - \Gamma_i}{\Delta_i} \right\rfloor$. The key is $\mathcal{K} = c_1 || \ldots || c_{50}$, and the template $\mathcal{T}$ is composed of $\{(\Delta_1, \Gamma_1), \ldots, (\Delta_{50}, \Gamma_{50})\}$. To later extract $\mathcal{K}$ given a biometric sample $\mathcal{B}'$, and a template $\mathcal{T}$, set $c'_i = \left\lfloor \frac{\phi_i(\mathcal{B}') - \Gamma_i}{\Delta_i} \right\rfloor$ and output $\mathcal{K}' = c'_1 || \ldots || c'_{50}$. We refer the reader to [41] for details on correctness.

As is the case in many other proposals, Vielhauer et al. perform an analysis that addresses requirement REQ-KR by arguing that given that the template leaks only error correcting information (i.e., the partitioning of the feature space) it does not indicate the values $c_i$. To support this argument, they conduct an empirical evaluation to measure the Shannon entropy of each $c_i$. For each user $u$ they derive $\mathcal{K}_u$ from $\mathcal{T}_u$ and $\mathcal{B}_u$, then compute the entropy of each element $c_i$ across all users. This analysis is a standard estimate of entropy. To see why this is inaccurate, consider two different users $a$ and $b$ such that $a$ outputs consistent values on feature $\phi$ and $b$ does not. Then the partitioning over $\phi$'s range differs for each user. Thus, even if the mean value of $\phi$ is the same when measured over both $a$'s and $b$'s samples, this mean will be mapped to different partitions in the feature space, and thus, a different key. *This implies that computing entropy over the $c_i$ overestimates security because the mapping induced by the templates artificially amplifies the entropy of the biometrics.* A more realistic estimate of the utility afforded an adversary by auxiliary information can be achieved by fixing a user's template, and using that template to error-correct every other user's samples to generate a list of keys, then measuring how close those keys are to the target user's key. By conditioning the estimate on the target users template we are able to eliminate the artificial inflation of entropy and provide a better estimate of the security afforded by the construction.

**Analysis.** We implemented the construction and tested the technique using all of the passphrases in the data set we collected in [3], which consists of over 9,000 writing samples from 47 users. Each user wrote the same five passphrases 10-20 times. In our analysis we follow the standard approach to isolate the entropy associated with the biometric: we compute various entropy measures using each user's rendering of the same passphrase [29, 5, 36] (this approach is justified as user selected passphrases are assumed to have low entropy). Tolerance values were set such that the approach achieved a False Reject Rate (FRR) of $0.1\%$ (as reported in [40]) and all outliers and samples from users who failed to enroll [24] were removed.

Figure 1 shows three different measures of key uncertainty. The first measure, denoted *Standard*, is the common measure of interpersonal variation as reported in the literature (e.g., [17, 7]) using the data from our experiments. Namely, if the key element $c_i$ has entropy $H_i$ across the entire population, then the entropy of the key space is computed as $H = \sum_{i=1}^{50} H_i$. We also show two estimates of guessing distance, the first ($\mathsf{GD}(\mathcal{U}, \mathcal{P})$, plotted as GD-P) does not take a target user's template into account and $\mathcal{P}$ is just the distribution over all other users's keys in the population (the techniques we use to compute these estimates are described in Appendix B). The second ($\mathsf{GD}(\mathcal{U}, \mathcal{P}[\mathcal{T}_u])$, plotted as GD-U) takes the user's template into account, computing $\mathcal{P}[\mathcal{T}_u]$ by taking the biometrics from all other users in the population, and generating keys using $\mathcal{T}_u$, then computing the distribution over these keys.

Figure 1 shows the CDF of the number of guesses that one would expect an adversary to make to find each user's key. There are several important points to take away from these results. The first is the common pitfalls associated with computing key entropy. The difference between $\mathsf{GD}(\mathcal{U}, \mathcal{P})$ and the standard measurement indicates that the standard measurement of entropy (43 bits in this case) is clearly misleading—under this view one would expect an adversary to make $2^{42}$ guesses on average before finding a key. However, from $\mathsf{GD}(\mathcal{U}, \mathcal{P})$ it is clear that an adversary can do substantially better than this. The difference in estimates is due to the fact that GD takes into account conditional information between features whereas a more standard measure does not.

The second point is the impact of a user's template on computing GD. We can see by examining $\mathsf{GD}(\mathcal{U}, \mathcal{P})$ that if we take the usual approach of just computing entropy over the keys, and ignore each user's template, we would assume only a small probability of guessing a key in fewer than $2^{21}$ attempts. On the other hand, since the templates reduce the possible key space for each user, the estimate $\mathsf{GD}(\mathcal{U}, \mathcal{P}[\mathcal{T}_u])$ provides a more realistic measurement. In fact, an adversary with access to population

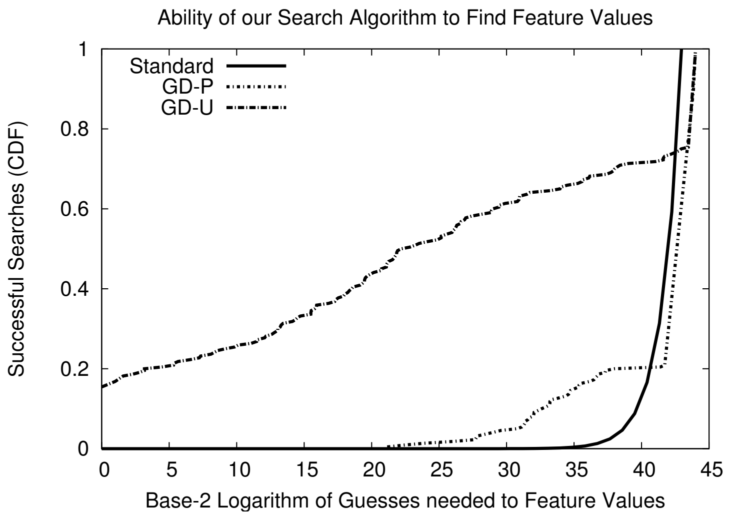


Figure 1: CDF of the guesses required by an adversary to find a key. We compare the *Standard* metric to two estimates of GD, one that uses the target user's template (GD-U), and one that uses each individual user's template (GD-P).

statistics has a $50\%$ chance of guessing a user's key in fewer than $2^{22}$ attempts, and $15.5\%$ chance guessing a key in a *single* attempt!

These results also shed light on another pitfall worth mentioning—namely, that of reporting an average case estimate of key strength. If we take the target user's template into account in the current construction, $15.5\%$ of the keys can be guessed in one attempt despite the estimated Guessing Entropy being approximately $2^{22}$. In summary, this analysis highlights the importance of conditioning entropy estimates on publicly available templates, and how several common entropy measures can result in misleading estimates of security.

## 6 The Impact of Public Information on Weak Biometric Privacy

Recall that a scheme that achieves Weak Biometric Privacy uses templates that do not leak information about the biometrics input during enrollment. A standard approach to arguing that a scheme achieves REQ-WBP is to show (1) auxiliary information leaks little useful information about the biometrics, and (2) templates do not leak information about a biometric. This can be problematic as the two steps are generally performed in isolation. In our description of REQ-WBP, however, we argue that step (2) should actually show that an adversary with access to *both* templates and auxiliary information should learn no information about the biometric. The key difference here is that auxiliary information is used in both steps (1) and (2). This is essential as it is not difficult to create templates that are secure when considered in isolation, but are insecure once we consider knowledge derived from *other* users (e.g., population-wide statistics). In what follows we shed light on this important consideration by examining the scheme of Hao and Wah [17]. While our analysis focuses on their construction, it is pertinent to any BKG that stores partial information about the biometric in the template [43, 26].

For completeness, we briefly review the construction. The BKG generates DSA signing keys from $n$ dynamic features associated with handwriting (e.g., pen tip velocity or writing time). The range of each feature is quantized based on a user's natural variation over the feature. Each partition of a feature's range is assigned a unique integer; let $p_i$ be the integer that corresponds to the partition containing the output of feature $\phi_i$ when applied to the user's biometric. The signing key is computed as $\mathcal{K} = \mathrm{SHA1}(p_1||\ldots||p_n)$. The template stores information that describes the partitions for each feature, as well as the $(x, y)$ coordinates that define the pen strokes of the enrollment samples, and the verification key corresponding to $\mathcal{K}$. The $(x, y)$ coordinates of the enrollment samples are used as input to the Dynamic Time Warping [32] algorithm during subsequent key generation; if the provided sample diverges too greatly from the original samples, it is immediately rejected and key generation aborted.

Hao et al. performed a typical analysis of REQ-WBP [17]. First, they compute the entropy of the features over the entire population of users to show that auxiliary information leaks little information that could be used to discern the biometric. Second, the

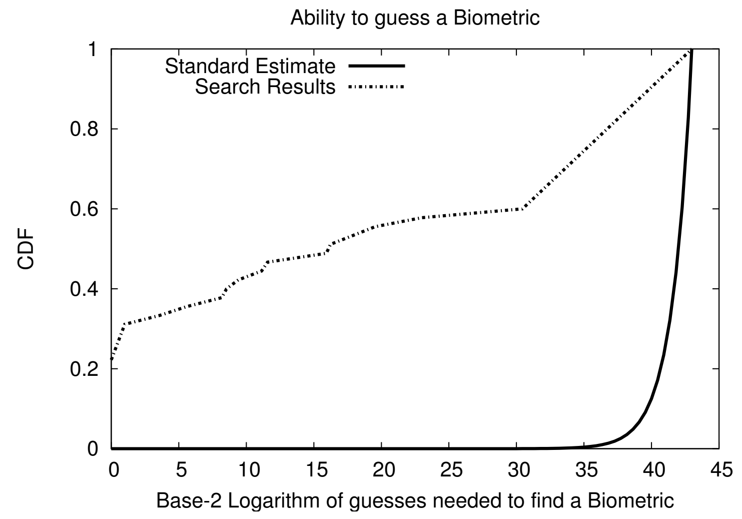


Figure 2: Search results against the BKG proposed by Hao et al. [17]. Our search algorithm has a 22% chance of finding a user's key on the first guess.

template is argued to be secure by making the following three observations. First, since the template only specifies the partitioning of the range of each feature, the template only leaks the variation in each feature, not the output. Second, for a computationally bound adversary, a DSA verification key leaks no information about the DSA signing key. Third, since the BKG employs only dynamic features, the static $(x, y)$ coordinates leak no relevant information. Note that in this analysis the template is analyzed *without* considering auxiliary information. Unfortunately, while by themselves the auxiliary information and the templates seem to be of little use to an adversary, when taken together, the biometric can be easily recovered.

**Analysis.** To demonstrate this, we apply the techniques of [3] to generate guesses of the user's biometric samples. In [3] we describe a set of statistical measures that can be computed using population statistics, and map these spatial measures [3] to the most likely pen speed. In that work we assume limited knowledge of the target user's biometric, and compose static samples from the user to create a partial forgery, then infer timing information to make a complete forgery. In the current approach, we need not assume access to the target user's biometric because the $(x, y)$ coordinates of the enrollment samples are stored in the template. Thus, we apply our approach from [3], to make a guess at the user's biometric. Then, we use an intelligent search algorithm that enumerates other biometrics that are "close" to the first guess. The algorithm focuses the bulk of its work searching for the outputs of the features that exhibit high variance across the population, and reduces the search space by exploiting conditioning between features.

To empirically evaluate our attack, we used the same data set as in Section 5. Our implementation of the BKG had a FRR of 29.2% and a False Accept Rate (FAR) of 1.7%, which is inline with the FRR/FAR of 28%/1.2% reported in [17]. Moreover, if we follow the computation of inter-personal variation as described in [17], then we would incorrectly conclude that the scheme creates keys with over 40 bits of entropy with our data set, which is the same estimate provided in [17]. However, this is not the case (see Figure 2). In particular, the fact that the template leaks information about the biometric enables an attack that successfully recreates the key 22% of the time on the *first* try; approximately 50% of the keys are correctly identified after making fewer than $2^{15}$ guesses. In summary, the significance of this analysis does not lie in the effectiveness of the described attack, but more so in the fact that the original analysis failed to take auxiliary information into consideration when evaluating the security of the template.

## 7 The Impact of Key Compromise on Strong Biometric Privacy

Lastly, we highlight the importance of quantifying the privacy of a user's biometric against adversaries who have access to the cryptographic key (i.e., REQ-SBP from Section 3). We examine a BKG proposed by Hao et al. [16].[4] The construction generates a random key and then "locks" it with a user's iris code. The construction uses a cryptographic hash function $h : \{0,1\}^* \rightarrow \{0,1\}^s$ and a "concatenated" error correction code consisting of an encoding algorithm $C : \{0,1\}^{140} \rightarrow \{0,1\}^{2048}$, and the corresponding decoding algorithm $D : \{0,1\}^{2048} \rightarrow \{0,1\}^{140}$. This error correction code is the composition of a Reed-Solomon and Hadamard code [16, Section 3]. Iris codes are elements in $\{0,1\}^{2048}$ [8].

The BKG works as follows: given a user's iris code $\mathcal{B}$, select a random string $\mathcal{K} \in \{0,1\}^{140}$, and derive the template $\mathcal{T} = \langle h(\mathcal{K}), \mathcal{B} \oplus C(\mathcal{K}) \rangle$, and output $\mathcal{T}$ and $\mathcal{K}$. To later derive the key given an iris code $\mathcal{B}'$ and the template $\mathcal{T} = \langle t_1, t_2 \rangle$, compute $\mathcal{K}' = D(t_2 \oplus \mathcal{B}')$. If $h(\mathcal{K}') = t_1$, then output $\mathcal{K}'$, otherwise, fail. If $\mathcal{B}$ and $\mathcal{B}'$ are "close" to one another, then $t_2 \oplus \mathcal{B}'$ is "close" to $C(\mathcal{K})$, perhaps differing in only a few bits. The error correcting code handles these errors, yielding $\mathcal{K}' = \mathcal{K}$.

Hao et al. provide a security analysis arguing requirement REQ-KR using both cryptographic reasoning and a standard estimate of entropy of the input biometric. That is, they provide empirical evidence that auxiliary information cannot be used to guess a target user's biometric, and a cryptographic argument that, assuming the former, the template and auxiliary information cannot be used to guess a key. They conservatively estimate the entropy of $\mathcal{K}$ to be 44 bits. Moreover, the authors note that if the key is ever compromised, the system can be used to "lock" a new key, since $\mathcal{K}$ is selected at random and is not a function of the biometric.

Unfortunately, given the current construction, compromise of $\mathcal{K}$, in addition to the public information $\mathcal{T} = \langle t_1, t_2 \rangle$, allows one to completely reconstruct $\mathcal{B} = C(\mathcal{K}) \oplus t_2$. Thus, even if a user were to create a new template and key pair, an adversary could use the old template and key to derive the biometric, and then use the biometric to unlock the new key. The significance of this is worth restating: because this BKG fails to meet REQ-SBP, the privacy of a user's biometric is completely undermined once any key for that user is ever compromised.

## 8 Conclusion

In this paper, we examine a series of requirements, pitfalls, and subtleties that are commonly overlooked in the evaluation of biometric key generators. Our goal is to encourage rigorous empirical evaluations that consider the impact of publicly available data to show that a BKG (*I.*) ensures the privacy of a user's biometric, and (*II.*) outputs keys that are suitable for cryptographic applications. Our exposition brings to the forefront *practical* ways of thinking about existing requirements that help elucidate subtle nuances that are commonly overlooked in regards to biometric security. As we demonstrate, failure to consider these requirements may result in estimates that overstate the security of proposed schemes.

To underscore the practical significance of each of these requirements, we present analyses of three published systems. While we point out weaknesses in specific constructions, it is not our goal to fault the those specific works. Instead, we aim to bring to light flaws in the standard approaches that were followed in each setting. In one case we exploit auxiliary information to show that an attacker can guess $15\%$ of the keys on her first attempt. In another case, we highlight the importance of ensuring biometric privacy by exploiting the information leaked by templates to yield a $22\%$ chance of guessing a user's key in one attempt. Lastly, we show that subtleties in BKG design can lead to flaws that allow an adversary to derive a user's biometric given a compromised key and template, thereby completely undermining the security of the scheme.

We hope that our work encourages designers and evaluators to analyze BKGs with a degree of skepticism, and to question claims of security that overlook the requirements presented herein. To facilitate this type of approach, we not only ensure that our requirements can be applied to real systems, but also introduce *Guessing Distance*—a heuristic measure that estimates the uncertainty of the outputs of a BKG given access to population statistics.

## Acknowledgements

We would like to thank Fabian Monrose for invaluable contributions to this work. We would also like to thank Dan Lopresti for providing helpful feedback on earlier versions of this paper. This research was funded in part by NSF Grant CNS-0430338.

## References

[1] ADLER, A. Images can be Regenerated from Quantized Biometric Match Score Data. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering* (Niagara Falls, Canada, May 2004), pp. 469–472.

[2] ALVARE, A. How crackers crack passwords or what passwords to avoid. In *Proceedings of the Second USENIX Security Workshop* (August 1990), pp. 103–112.

[3] BALLARD, L., LOPRESTI, D., AND MONROSE, F. Evaluating the security of handwriting biometrics. In *The $10^{th}$ International Workshop on the Foundations of Handwriting Recognition* (October 2006), pp. 461–466.

[4] BALLARD, L., LOPRESTI, D., AND MONROSE, F. Forgery quality and its implications for behavioral biometric security. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Special Edition) 37*, 5 (October 2007), 1107–1118.

[5] BALLARD, L., MONROSE, F., AND LOPRESTI, D. Biometric authentication revisited: Understanding the impact of wolves in

sheep's clothing. In *Proceedings of the 15th Annual Usenix Security Symposium* (Vancouver, BC, Canada, August 2006), pp. 29–41.

[6] BOYEN, X. Reusable cryptographic fuzzy extractors. In *ACM Conference on Computer and Communications Security—CCS 2004* (2004), New-York: ACM Press, pp. 82–91.

[7] CHANG, Y.-J., ZHANG, W., AND CHEN, T. Biometrics-Based Cryptographic Key Generation. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME)* (2004), vol. 3, pp. 2203–2206.

[8] DAUGMAN, J. The importance of being random: Statistical principles of iris recognition. *Pattern Recognition 36* (2003), 279–291.

[9] DAVIDA, G. I., FRANKEL, Y., AND MATT, B. J. On enabling secure applications through off-line biometric identification. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (May 1998), pp. 148–157.

[10] DAVIS, D., MONROSE, F., AND REITER, M. K. On user choice in graphical password schemes. In *Proceedings of the 13th USENIX Security Symposium* (August 2004), pp. 151–164.

[11] DODIS, Y., REYZIN, L., AND SMITH, A. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in Cryptology - EUROCRYPT 2004* (2005), vol. 3027 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 523–540. Full version appears as Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data, *IACR ePrint Cryptography Archive 2003/235*.

[12] DODIS, Y., AND SMITH, A. Correcting errors without leaking partial information. In *Proc. 37th ACM Symp. on Theory of Computing* (2005), ACM, pp. 654–663.

[13] DOLE, B., LODIN, S., AND SPAFFORD, E. Misplaced trust: Kerberos 4 session keys. In *Proceedings of the 1997 Symposium on Network and Distributed System Security* (Washington, DC, USA, 1997), IEEE Computer Society, p. 60.

[14] FELDMEIER, D., AND KARN, P. UNIX password security – ten years later. In *Advances in Cryptology – CRYPTO '89 Proceedings* (Berlin, Germany, 1990), vol. 435 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 44–63.

[15] GOH, A., AND NGO, D. C. L. Computation of cryptographic keys from face biometrics. In *Proceedings of Communications and Multimedia Security* (2003), pp. 1–13.

[16] HAO, F., ANDERSON, R., AND DAUGMAN, J. Combining cryptography with biometrics effectively. *IEEE Transactions on Computers* (2006).

[17] HAO, F., AND WAH, C. Private key generation from on-line handwritten signatures. *Information Management and Computer Security 10*, 4 (2002), 159–164.

[18] HASTAD, J., IMPAGLIAZZO, R., LEVIN, L., AND LUBY, M. A pseudorandom generator from any one-way function. *SIAM Journal on Computing 28* (1998).

[19] JAIN, A. K., ROSS, A., AND ULUDAG, U. Biometric Template Security: Challenges and Solutions. In *Proceedings of European Signal Processing Conference (EUSIPCO)* (September 2005).

[20] JUELS, A., AND SUDAN, M. A fuzzy vault scheme. In *IEEE International Symposium on Information Theory* (2002).

[21] JUELS, A., AND WATTENBERG, M. A fuzzy commitment scheme. In *Proceedings of the 6th ACM Conference on Computer and Communication Security* (November 1999), pp. 28–36.

[22] LI, Q., SUTCU, Y., AND MEMON, N. Secure sketch for biometric templates. In *In Proceedings of Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security* (2006), pp. 99–113.

[23] LOPRESTI, D. P., AND RAIM, J. D. The effectiveness of generative attacks on an online handwriting biometric. In *Proceedings of the International Conference on Audio- and Video-based Biometric Person Authentication*. Hilton Rye Town, NY, USA, 2005, pp. 1090–1099.

[24] MANSFIELD, A. J., AND WAYMAN, J. L. Best practices in testing and reporting performance of biometric devices. Tech. Rep. NPL Report CMSC 14/02, Centre for Mathematics and Scientific Computing, National Physical Laboratory, August 2002.

[25] MASSEY, J. L. Guessing and entropy. In *Proceedings of the 1994 IEEE International Symposium on Information Theory* (1994), p. 204.

[26] MOHANTY, P., SARKAR, S., AND KASTURI, R. A non-iterative approach to reconstruct face templates from match scores. In *18th International Conference on Pattern Recognition (ICPR 2006)* (August 2006), pp. 598–601.

[27] MONROSE, F., REITER, M., LI, Q., LOPRESTI, D., AND SHIH, C. Towards speech-generated cryptographic keys on resource-constrained devices. In *Proceedings of the Eleventh USENIX Security Symposium* (2002), pp. 283–296.

[28] MONROSE, F., REITER, M. K., LI, Q., AND WETZEL, S. Cryptographic key generation from voice (extended abstract). In *Proceeedings of the 2001 IEEE Symposium on Security and Privacy* (May 2001), pp. 12–25.

[29] MONROSE, F., REITER, M. K., AND WETZEL, S. Password hardening based on keystroke dynamics. *International Journal of Information Security 1*, 2 (February 2002), 69–83.

[30] NISAN, N., AND ZUCKERMAN, D. Randomness is linear in space. *Journal of Computer and Systems Science 52*, 1 (1996), 43–52.

[31] RATHA, N. K., CONNELL, J. H., AND BOLLE, R. M. Enhancing security and privacy in biometrics-based authentication systems. *IBM Syst. J. 40*, 3 (2001), 614–634.

[32] SANKOFF, D., AND KRUSKAL, J. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, second ed. Addison-Wesley Publishing, Reading, MA, 1999.

[33] SOUTAR, C., ROBERGE, D., STOIANOV, A., GILROY, R., AND KUMAR, B. V. Biometric encryption$^{tm}$ using image processing. In *Optical Security and Counterfeit Deterrence Techniques II* (1998), vol. 3314, IS&T/SPIE, pp. 178–188.

[34] SOUTAR, C., AND TOMKO, G. J. Secure private key generation using a fingerprint. In *Cardtech/Securetech Conference Proceedings* (May 1996), pp. 245–252.

[35] SUTCU, Y., SENCAR, H. T., AND MEMON, N. A Secure Biometric Authentication Scheme based on Robust Hashing. In *Proceedings of the 7th Workshop on Multimedia and Security* (New York, NY, USA, 2005), pp. 111–116.

[36] THORPE, J., AND VAN OORSCHOT, P. Human-Seeded Attacks and Exploiting Hot-Spots in Graphical Passwords. In *Proceedings of the 16th Annual Usenix Security Symposium* (Boston, MA, August 2007).

[37] ULUDAG, U., AND JAIN, A. Attacks on biometric systems: A case study in fingerprints. In *Proceedings of SPIE-EI 2004, Security, Steganography and Watermarking of Multimedia Contents VI*, vol. 5306, pp. 622–633.

[38] ULUDAG, U., AND JAIN, A. Securing fingerprint template: Fuzzy vault with helper data. In *Proceedings of the IEEE Workshop on Privacy Research In Vision (PRIV)* (New York, NY, June 2006).

[39] ULUDAG, U., PANKANTI, S., PRABHAKAR, S., AND JAIN, A. K. Biometric cryptosystems: Issues and challenges. *Proceedings of the IEEE: Special Issue on Multimedia Security of Digital Rights Management 92*, 6 (2004), 948–960.

[40] VIELHAUER, C., AND STEINMETZ, R. Handwriting: Feature correlation analysis for biometric hashes. *EURASIP Journal on Applied Signal Processing 4* (2004), 542–558.

[41] VIELHAUER, C., STEINMETZ, R., AND MAYERHOFER, A. Biometric hash based on statistical features of online signatures. In *Proceedings of the Sixteenth International Conference on Pattern Recognition* (2002), vol. 1, pp. 123–126.

[42] WAYMAN, J. Fundamentals of biometric authentication technologies. *International Journal of Image & Graphics 1*, 1 (January 2001), 93–114.

[43] YAMAZAKI, Y., NAKASHIMA, A., TASAKA, K., AND KOMATSU, N. A study on vulnerability in on-line writer verification system. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition* (Seoul, South Korea, August-September 2005), pp. 640–644.

[44] ZHANG, W., CHANG, Y.-J., AND CHEN, T. Optimal thresholding for key generation based on biometrics. In *Proceedings of the International Conference on Image Processing (ICIP04)* (2004), vol. 5, pp. 3451–3454.

[45] ZHENG, G., LI, W., AND ZHAN, C. Cryptographic key generation from biometric data using lattice mapping. In *ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 513–516.

## Notes

[1]Typically, $\mathcal{U}$ is computed over error-corrected values, and so the most likely element will also be the only element that has any probability mass.

[2]We note that Guessing Distance is not a distance metric as it does not necessarily satisfy symmetry or the triangle inequality.

[3]These measures can be reproduced given the $(x, y)$ coordinates of handwriting.

[4]This BKG is technically an instance of the fuzzy commitment proposed by Juels and Wattenberg [21], which was later shown to be an instance of a secure sketch [11].

## A Guessing Distance and Guessing Entropy

Guessing Entropy [25] is commonly used for measuring the expected number of guesses it takes to find an average element in a set assuming an optimal guessing strategy (i.e., first guessing the element with the highest likelihood, followed by guessing the element with the second highest likelihood, etc.). Given a distribution $\mathcal{P}$ over $\Omega$ and the convention that $\mathcal{P}(\omega_i) \geq \mathcal{P}(\omega_{i+1})$, Guessing Entropy is computed as $G(\mathcal{P}) = \sum_{i=1}^{n} i\mathcal{P}(\omega_i)$.

Guessing Entropy is commonly used to determine how many guesses an adversary will take to guess a key. At first, Guessing Entropy and Guessing Distance appear to be quite similar. However, there is one important difference: Guessing Entropy is a summary statistic and Guessing Distance is not. While Guessing Entropy provides an intuitive and accurate estimate over distributions that are close to uniform, the fact that there is one measure of strength for all users in the population may result in somewhat misleading results when Guessing Entropy is computed over skewed distributions.

To see why this is the case, consider the following distribution: let $\mathcal{P}$ be defined over $\Omega = \{\omega_1, \dots, \omega_n\}$ as $\mathcal{P}(\omega_1) = \frac{1}{2}$, and $\mathcal{P}(\omega_i) = \frac{1}{2(n-1)}$ for $i \in [2, n]$. That is, one element (or key) is output by 50% of the users and the remaining elements are output with equal likelihood. The Guessing Entropy of $\mathcal{P}$ is:

$$\begin{aligned} G(\mathcal{P}) &= \sum_{i=1}^{n} i\mathcal{P}(\omega_i) \\ &= \mathcal{P}(\omega_1) + \sum_{i=2}^{n} i\mathcal{P}(\omega_i) \\ &= \frac{1}{2} + \frac{1}{2(n-1)} \sum_{i=2}^{n} i \\ &= \frac{1}{2} + \frac{1}{2(n-1)} \left( \frac{n(n+1)}{2} - 1 \right) \\ &\approx \frac{n}{4} \end{aligned}$$

Thus, although the expected number of guesses to correctly select $\omega$ is approximately $\frac{n}{4}$, over half of the population's keys are correctly guessed on the first attempt following the optimal strategy. To contrast this, consider an analysis of Guessing Distance with threshold $\delta = \frac{1}{N}$. (Assume for exposition that distributions are estimated from a population of $N = 2(n-1)$ users.) To do so, evaluate each user in the population independently. Given a population of users, first remove a user to compute $\mathcal{U}$ and use the remaining users to compute $\mathcal{P}$. Repeat this process for the entire population.

In the case of our pathological distribution, we may consider only two users without loss of generality: a user with distribution $\mathcal{U}_1$ who outputs key $\omega_1$, and user with distribution $\mathcal{U}_2$ who outputs key $\omega_2$. In the first case, we have $\mathsf{GD}_\delta(\mathcal{U}_1, \mathcal{P}) = \log 1 = 0$, because the majority of the mass according to $\mathcal{P}$ is assigned to $\omega_1$, which is the most likely element according to $\mathcal{U}_1$. For $\mathcal{U}_2$, we have $t^- = 2$ and $t^+ = n$, and thus $\mathsf{GD}_\delta(\mathcal{U}_2, \mathcal{P}) = \log \frac{n+2}{2}$. Taking the minimum value (or even reporting a CDF) shows that for a large proportion of the population (all users with distribution $\mathcal{U}_1$), this distribution offers no security—a fact that is immediately lost if we only consider a summary statistic. However, it is comforting to note, that if we compute the average of $2^{\mathsf{GD}}$ over all users, we obtain estimates that are identical to that of guessing entropy for sets that are sufficiently large:

$$\begin{aligned}\frac{1}{N}\sum_{(\mathcal{U},\mathcal{P})} 2^{\mathsf{GD}_\delta(\mathcal{U},\mathcal{P})} &= \frac{1}{N}\left(\frac{N}{2}2^{\log 1}+\frac{N}{2}2^{\log\frac{n+2}{2}}\right)\\ &= \frac{1}{2}+\frac{1}{2}\left(\frac{n+2}{2}\right)\\ &\approx \frac{n}{4}\end{aligned}$$

## B Estimating GD

As noted in Section 5, it is difficult to obtain a meaningful estimate of probability distributions over large sets, e.g., $\mathbb{N}^{50}$. In order to quantify the security defined by a system, it is necessary to find techniques to derive meaningful estimates. This Appendix discusses how we estimate GD. The estimate also implicitely defines an algorithm that can be used to guess keys.

For convenience we use $\phi$ to denote both a biometric feature and the random variable that is defined using population statistics over $\phi$ (taken over the set $\Omega_\phi$). If a distribution is not subscripted, it is understood to be taken over the key space $\Omega = \Omega_{\phi_1}\times\cdots\times\Omega_{\phi_n}$. Our estimate uses of several tools from information theory:

**Entropy.** The entropy of a random variable $X$ defined over the set $\Omega$ is

$$H(X) = -\sum_{\omega\in\Omega}\Pr[X=\omega]\log\Pr[X=\omega]$$

**Mutual Information.** The amount of information shared between two random variables $X$ and $Y$ defined over the domains $\Omega_X$ and $\Omega_Y$ is measured as

$$I(X,Y) = \sum_{x\in\Omega_x}\sum_{y\in\Omega_y}\Pr[X=x\wedge Y=y]\log\frac{\Pr[X=x\wedge Y=y]}{\Pr[X=x]\Pr[Y=y]}$$

We use the notation $I(X;Y,Z)$ to denote the mutual information between the random variable $X$ and the random variable defined by the joint distribution between the random variables $Y$ and $Z$.

**The Estimate.** Let $\mathsf{GD}_\delta(\mathcal{U}_{\phi_i},\mathcal{P}_{\phi_i}|u_{i-1},\ldots,u_1)$ be the guessing distance between the user's and population's distribution over $\phi_i$ conditioned on the even that $\phi_{i-1}=u_{i-1},\ \ldots,\ \phi_1=u_1$. In particular, let $L_{\mathcal{P}_{\phi_i}}=(\omega_1,\ldots,\omega_n)$ be the elements of $\Omega_{\phi_i}$ ordered such that

$$\mathcal{P}_{\phi_i}(\omega_j|\phi_{i-1}=u_{i-1},\ \ldots,\ \phi_1=u_1)\geq \mathcal{P}_{\phi_i}(\omega_{j+1}|\phi_{i-1}=u_{i-1},\ \ldots,\ \phi_1=u_1)$$

As before, let $\omega^*=\mathrm{argmax}_{\omega\in\Omega_{\phi_i}}\mathcal{U}_{\phi_i}(\omega)$, and $t^-$ and $t^+$ be the smallest and largest indexes $j$ such that

$$|\mathcal{P}_{\phi_i}(\omega_j|\phi_{i-1}=u_{i-1},\ \ldots,\ \phi_1=u_1) - \mathcal{P}_{\phi_i}(\omega^*|\phi_{i-1}=u_{i-1},\ \ldots,\ \phi_1=u_1)|\leq\delta$$

Then, $\mathsf{GD}_\delta(\mathcal{U}_{\phi_i},\mathcal{P}_{\phi_i}\mid u_{i-1},\ldots,u_1)=\log(t^-+t^+)-1$. In other words, if an adversary assumes that a target user is distributed according to the population and fixes the values of certain features, this is the number of guesses she will need to make to guess another feature. Unfortunately, this quantity is also infeasible to compute in light of data constraints so we endeavor to find an easily computable estimate. To this end, define the weight $(d_i)$ of an element in $\omega\in\Omega_{\phi_i}$ as:

$$d_i(\omega\mid u_{i-1},\ldots,u_1) = \sum_{h=1}^{i-1}\sum_{j=1}^{i-1} I(\phi_i;\phi_h,\phi_j)\,\mathcal{P}_{\phi_i}(\omega\mid\phi_h=u_h\wedge\phi_j=u_j)$$

The weights of elements that are more likely to occur given the values of other features will be larger than the weights that are less likely to occur. Intuitively, each of the values $(u)$ has an influence on $d_i(\omega)$ and those values that correspond to features that have a higher correlation with $\phi_i$ have more influence. We also note that we only use two levels of conditional probabilities, which are relatively easy to compute, instead of conditioning over the entire space. Now, we use the weights to estimate the probability distributions as:

$$\hat{\mathcal{P}}_{\phi_i}(\omega_j\mid u_{i-1},\ldots,u_1) = d_i(\omega_j\mid u_{i-1},\ldots,u_1)/\sum_{\omega\in\Omega_{\phi_i}} d_i(\omega\mid u_{i-1},\ldots,u_1)$$

Note that while this technique may not provide a perfect estimate of each probability, our goal is to discover the relative magnitude of the probabilities because they will be used to estimate Guessing Distance. We believe that this approach achieves this goal.

We are almost ready to provide an estimate of GD. First, we specify an ordering for the features. The ordering will be according to an ordering measure $(M(\phi))$ such that features with a larger measure have a low entropy (and are therefore easier to guess) and have a high correlation with other features. An adversary could then use this ordering to reduce the number of guesses in a search by first guessing features with a higher measure. Define the feature-ordering measure for $\phi_i$ as:

$$M(\phi_i)=\sum_{i\neq j}\left(1+\frac{H(\phi_j)}{H(\phi_i)}\right)^{1+I(\phi_i,\phi_j)}$$

Finally, we reindex the features such that $M(\phi_i) \geq M(\phi_{i+1})$ for all $i \in [1, 50]$, and estimate the guessing distance for a specific user with $\phi_i = x_i$ as:

$$\widehat{\mathsf{GD}}(\mathcal{U}, \mathcal{P}) =$$
$$\log\left(1+\sum_{i=1}^{50}\left(\left(2^{\mathsf{GD}(\mathcal{U}_{\phi_i}, \hat{\mathcal{P}}_{\phi_i}|x_{i-1},\ldots,x_1)}-1\right)\prod_{j=i+1}^{50}|\Omega_{\phi_j}|\right)\right)$$

This estimate helps in modeling an adversary that performs a brute-force search over all of the features by starting with the features that are easiest to guess and using those features to reduce the uncertainty about features that are more difficult to guess. For each feature, the adversary will need to make $2^{\mathsf{GD}(\mathcal{U}_{\phi_i}, \hat{\mathcal{P}}_{\phi_i}|u_{i-1},\ldots,u_1)}$ guesses to find the correct value. Since each incorrect guess ($2^{\mathsf{GD}(\mathcal{U}_{\phi_i}, \hat{\mathcal{P}}_{\phi_i}|u_{i-1},\ldots,u_1)} - 1$ of them) will cause a fruitless enumeration of the rest of the features, we multiply the number of incorrect guesses by the sizes of the ranges of the remaining features. Finally, we take the log to represent the number of guesses as bits.

Section 5 uses this estimation technique to measure $\mathsf{GD}$ of a user versus the population ($\widehat{\mathsf{GD}}(\mathcal{U}, \mathcal{P})$), and for a user versus the population conditioned on the user's template ($\widehat{\mathsf{GD}}(\mathcal{U}, \mathcal{P}[\mathcal{T}_u])$). The only way in which the estimation technique differs between the two settings is the definition of $\mathcal{P}_{\phi_i}$. In the case of $\widehat{\mathsf{GD}}(\mathcal{U}, \mathcal{P})$, $\mathcal{P}_{\phi_i}$ is computed by measuring the $i^{\text{th}}$ key element for every other user in the population. In the case of $\widehat{\mathsf{GD}}(\mathcal{U}, \mathcal{P}[\mathcal{T}_u])$, $\mathcal{P}_{\phi_i}$ is computed using all of the other user's samples in conjuction with the target user's template to derive a set of keys and taking the distribution over the $i$th element of the keys.

# Unidirectional Key Distribution Across Time and Space with Applications to RFID Security

Ari Juels
*RSA Laboratories*
*Bedford, MA, USA*
*ajuels@rsa.com*

Ravikanth Pappu
*ThingMagic Inc*
*Cambridge, MA, USA*
*ravi.pappu@thingmagic.com*

Bryan Parno
*Carnegie Mellon University*
*Pittsburgh, PA, USA*
*parno@cmu.edu*

## Abstract

We explore the problem of secret-key distribution in *unidirectional* channels, those in which a sender transmits information blindly to a receiver. We consider two approaches: (1) Key sharing across *space*, i.e., via simultaneously emitted values that may follow different data paths and (2) Key sharing across *time*, i.e., in temporally staggered emissions. Our constructions are of general interest, treating, for instance, the basic problem of constructing highly compact secret shares. Our main motivating problem, however, is practical key management in RFID (Radio-Frequency IDentification) systems. We describe the application of our techniques to RFID-enabled supply chains and a prototype privacy-enhancing system.

## 1 Introduction

Key management is a cornerstone of cryptography, but also its major deployment challenge. Textbook cryptographic protocols often presuppose keys held by a pair of principals anecdotally dubbed Alice and Bob. From birth, Alice and Bob are presumed to share a password, a secret key, or the public key of some mutually trusted entity.

In practice, the conceptually simple goals of key distribution—even between two parties—are fraught with complexity. Disparate naming conventions and requirements for key revocation and recovery have hobbled many public-key infrastructures. Password management remains a widespread challenge thanks to obstacles as varied as limited human memory, caps-lock keys, and social-engineering attacks such as phishing.

Ultimately, key distribution must rely on secure channels established through pre-existing trust relationships or special physical considerations. For example, browser software shipped with new computing systems carries the root public keys of a number of certificate authorities. Special physical assumptions and adversarial constraints can shape the problem of key distribution in interesting ways. Researchers have explored various physical models to support key establishment between pairs of devices, including optical channels [16,24], distance-bounding [30] based on signal velocity, and physical contact [33]. Such models treat a variety of adversarial capabilities. For instance, privacy amplification [3], which strengthens keys using shared sources of noise or quantum phenomena, appeals to bounds on adversarial data access or storage.

In this paper, we focus on the problem of key distribution between two parties communicating via a *unidirectional* channel. This special constraint means that one party (Alice) acts exclusively as a sender, while the other (Bob) acts exclusively as a receiver. We consider the challenge of unidirectional key transport when Alice and Bob have no pre-existing relationship, but share a channel with limited adversarial access. We believe that such special unidirectional models have broad applicability, as they reflect the natural broadcast characteristics of many media. The starting point and motivation for our investigation, though, is the specific, real-world problem of key transport in RFID-enabled supply chains.

**Organization** In Section 2, we give details on the RFID challenges motivating our work. We provide an overview of our technical contributions in Section 3 and review related work in Section 4. In Section 5, we present what we call *secret sharing in space*, a key-distribution system that supports privacy protection in RFID applications. We also briefly describe a prototype RFID implementation of secret sharing in space. In Section 6, we present *secret sharing in time*, a separate body of techniques applicable to RFID access-control and authentication, and also of broad interest for key distribution in unidirectional channels. We conclude in Section 7 with a brief discussion of future research directions.

## 2 Motivation: The RFID Landscape

The ratio of terrestrial radio and cellular telephone systems to the number of humans on earth is approaching unity, and in the past decade, a completely different kind of radio device has emerged and is poised to eclipse this ratio by three orders of magnitude. Rapid advances in CMOS technology have enabled the production of low-cost *tags* that are capable of reporting their identity over a wireless link. These tags—usually costing tens of cents and carrying a few thousand gates of silicon—have little if any general-purpose computing power beyond what is needed to respond to commands from an interrogator or *reader*. This asymmetry between interrogators and tags is further amplified by the fact that, in many applications, tags are passive, lacking an on-board source of power; instead, they harvest power from the electric, magnetic or electromagnetic field generated by the interrogators.

Recent developments in passive Radio Frequency IDentification (RFID) technology and corresponding international standards [12] have spurred deployment in applications ranging from supply-chain and inventory management of consumer goods, to tracking medical equipment in hospitals, to counting poker chips on gaming tables.

The heir apparent to the optical barcode, RFID is becoming a prevalent technology in supply-chain management. Ultimately, manufacturers and retailers envisage RFID tagging of individual consumer *items*. Today, tagging is most common at the granularity of *cases*, which contain consumer items, and of *pallets*, which carry cases. In this paper, we use the term "case" as the generic term for a discrete collection of goods.

For supply-chain operations, the predominant RFID standard is one known as the Electronic Product Code (EPC) (in particular, Class-1 Gen-2 EPC, hereafter referred to as Gen2). EPC tags act effectively as wireless barcodes, emitting short strings of information known as EPC codes. An EPC code has four basic components: (1) A *header*, which denotes the EPC version number; (2) A *domain manager*, which typically specifies the manufacturer or creator of the item; (3) An *object class*, which specifies the item type, and (4) a *serial number*, a unique identifier for the item. This *license plate* approach associates an arbitrary amount of metadata with the tagged object while requiring little memory on the tag itself.

### 2.1 Security and Key Distribution in Gen2

Two features in the Gen2 standard require secret keys:

**Locking and perma-locking:** It is possible to lock part (or all) of the tag's memory, either temporarily under a 32-bit password, or permanently with no possibility of unlocking and rewriting the memory. While this feature prevents unauthorized entities from tampering with the contents of tag memory, it does not prevent unauthorized readers from reading the contents.

**The kill command:** The only security function that completely disables tags is a command known as *kill*. When transmitted by a reader along with a tag-specific kill PIN (32 bits long in Gen2), the kill command causes a tag to disable itself permanently.

The EPC kill function is envisaged as a privacy-enhancing feature for retail environments with item-level tagging. EPC tags specify the items to which they are affixed. Thus a consumer carrying EPC-tagged items would in principle be subject to clandestine inventorying attacks that disclose sensitive data about medications, reading materials, luxury goods, and so forth. By deploying the kill function at the point of sale, a retail shop can protect against such privacy infringements by disabling tags. Additionally, researchers have proposed anti-cloning techniques that co-opt the kill and write-access commands in EPC to support reader authentication of tags and to protect PINs from untrusted readers [15].

Both locking and killing pose a significant implementation hurdle: They require a solution to the *key-distribution* problem. The initialization of tag-specific kill PINs in tags and the secure propagation of these PINs to point-of-sale devices are formidable operational challenges. Supply chains include entities with widely disparate data-processing capabilities. Information transfer across organizational boundaries, moreover, introduces a host of regulatory and technical burdens. Hence supply-chain entities commonly lack data-network mechanisms for timely, reliable, and secure transport of PINs. While it might seem a straightforward matter for Alice (a manufacturer) to share EPC PINs with Bob (a retailer) through a data network, in practice it is often quite difficult. Indeed, with all of the intermediaries through which manufactured goods regularly pass, Alice may even ship cases without knowing that Bob is the ultimate receiver.

In this paper, we show that RFID-enabled supply chains possess unique properties that allow us to:

- Provide consumer privacy with respect to unauthorized scanning of tagged objects;
- Provide a robust protocol-independent mechanism to distribute PINs and passwords *without requiring a network connection, changes to the air interface protocol, or changes to the tag hardware*.

The only resource our method requires is memory on the tag, and we provide a means to trade-off memory usage against security.

### 2.2 Object Hierarchies in RFID-Enabled Supply Chains

Our techniques for key distribution in RFID applications rely in part on the fact that supply chains are hierarchical in nature. To highlight the properties we utilize, we use Figure 1 to trace the path of a single pack of razor-blades in a consumer's home back to the manufacturing facility.

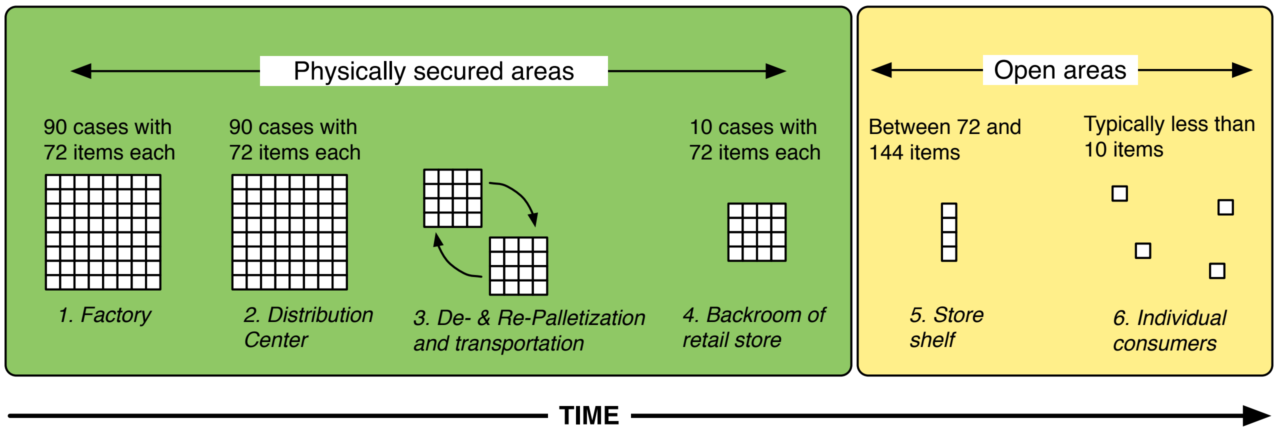

Physically secured areas
90 cases with 72 items each
90 cases with 72 items each
10 cases with 72 items each
1. Factory
2. Distribution Center
3. De- & Re-Palletization and transportation
4. Backroom of retail store
Open areas
Between 72 and 144 items
Typically less than 10 items
5. Store shelf
6. Individual consumers
TIME

Figure 1: **Object hierarchies in RFID-enabled supply chains** *This schematic represents the path taken by an individual pack of razor blades from the factory to the consumer's home. Please refer to Section 2.2 for details.*

Typically, items start off in large collections and progressively get whittled down into smaller aggregates as they make their way from the factory to the store shelf [13]. In the example above, razor blades are assembled into a pallet containing 90 cases, each with 72 packs of blades. Assuming the items, cases, and pallet are tagged, we have a total of 6571 tags on this particular pallet. The pallet is then transported, possibly with many other pallets, to a distribution center (DC). The DC de-palletizes the large pallet and assembles a mixed pallet with a smaller quantity of cases that has been ordered by the store. A typical number of cases from the original pallet that make it onto this new pallet is 10 [13]. Assuming a new pallet tag is added, 730 of the 6571 original tags are now available on the new pallet. This new pallet is then transported to the store and stored in the backroom. Of these 730 tags, typically up to two cases' worth, or 144, items are laid out on the store shelf for customers. From this collection, consumers pick up a few packs and purchase them. Therefore, the object hierarchy is as follows.

*Razor blades*: $6571 \rightarrow 730 \rightarrow 144 \rightarrow 5$

Similarly, for DVDs a typical object hierarchy is

*DVDs*: $5040 \rightarrow 2520 \rightarrow 400 \rightarrow 24$

where the last number represents an estimate of the number of DVDs from a case sold to an individual consumer. Finally, for pharmaceuticals, we have

*Pharmaceuticals*: $7200 \rightarrow 1920 \rightarrow 150 \rightarrow 6$

where again the last number represents an estimate of the maximum number of filled prescriptions from one case in possession of a consumer at the same time.

While these numbers may vary between different types of retailers and use cases, the important point to note is that the number of tagged items starts off large and ends up being small. Another important insight is that larger numbers of tags are typically found in physically secure areas, while smaller numbers of tags are found in physical locations that are accessible to adversaries. We exploit the fact that tags share the same space-time context earlier in the supply chain, but this history is progressively lost as tagged objects emerge from the supply chain into the front of the retail store and thereon into the consumer's home.

## 3 Our Contribution

The challenges of EPC PIN distribution motivate us to consider a new approach, that of *transporting secret keys in RFID tags themselves*. This approach allows a unidirectional model of key transport. The sender (Alice) encodes secrets across tags or cases. The receiver (Bob) recovers these secrets without communicating with Alice—and, potentially, without even knowing her identity.

To support this unidirectional model of key transport, we propose protocols for dispersing keys or PINs across tags by means of *secret sharing*. We consider two distinct modes of secret sharing: (1) *Secret sharing across space* and (2) *Secret sharing across time*.

**Secret sharing across space:** Alice can share a secret key $\kappa$ across a set of tags $T = \{\tau_1, \ldots, \tau_n\}$ in a case. To do so, she transforms $\kappa$ into a collection of shares $S_1, \ldots, S_n$, and stores $S_i$ on tag $\tau_i$, such that $\kappa$ can only be recovered by scanning all $n$ tags in the cases. (We later consider threshold secret sharing, i.e., schemes such that $k < n$ shares suffice for recovery of $\kappa$.)

Such secret sharing across tags permits a new approach to privacy enforcement for item-level tagging that largely *eliminates the need for killing tags*. Suppose that $m_i$ consists of the data, e.g., EPC code, associated with tag $\tau_i$. Suppose that Alice replaces $m_i$ with $E_\kappa[m_i]$ in all tags, where $E_\kappa$ represents symmetric-key encryption under $\kappa$. Then the contents $m_i$ of any tag can only be deciphered by scanning the full set of tags $T$.

On receiving a case from Alice, a retailer (Bob) can recover $\kappa$ and decrypt the EPC codes in its tags. *Once the items and their associated tags are dispersed by sale to customers, however, a would-be eavesdropper has no practical way to recover* $\kappa$. We assume here that access to tags is secured in the supply chain, i.e., the pre-sale environment. We illustrate the principle by example.

**Example 1** *Alice ships a case containing three bottles of medicine bearing RFID tags* $\tau_1, \tau_2$ *and* $\tau_3$ *with data strings* $m_1, m_2$*, and* $m_3$*. She generates a secret key* $\kappa$ *and transforms it into a triplet of shares* $(S_1, S_2, S_3)$ *via a (3,3)-secret sharing scheme. Alice writes the value* $v_i = (E_\kappa[m_i], S_i)$ *to tag* $\tau_i$*.*

*Bob, a pharmacist, receives Alice's case. He scans the three tags, recovers* $\kappa$ *and decrypts the data strings of the tags in the cases, enabling him to read* $m_1$ = *"High street-value drug, 500 mg, 100 count, bottle #8278732," as well as* $m_2$ *and* $m_3$*. Bob dispenses the first bottle to Carol.*

*Later in the day, a drug thief surreptitiously scans Carol's RFID tags as she passes on the street. The thief obtains the value* $v_1 = (E_\kappa[m_1], S_1)$*—a ciphertext and key share that by themselves carry no meaning and therefore do not reveal the presence of high-value pharmaceuticals.*

As this example illustrates, Bob does not have to perform any explicit action to protect his customers' privacy. He does not have to kill or rewrite tags. Secret sharing across space enforces privacy implicitly through the physical dispersion of tags. Unlike killing, though, secret sharing does not enforce privacy against tracking attacks. The value $v_1$ is itself a unique identifier that can serve to correlate different instances of scanning of Carol's tags and potentially track Carol herself. This is a basic limitation of our scheme, but one we consider to be of considerably smaller importance than revelation of tag data contents.

Of course, it is possible to encode $\kappa$ in a case-specific tag, rather than across items within a case. The advantage of sharing across space is twofold, though: (1) As we show, it allows for robust secret recovery, i.e., recovery of $\kappa$ even in the face of scanning errors or lost data and (2) It eliminates the need for an extra tag, i.e., one on each case.

Our main research challenge in applying secret sharing across space to RFID is the development of schemes with *tiny* secret shares. While the literature on computational secret sharing considers shares of length equal to that of a secret key, e.g., 128 bits, space constraints on EPC tags urge even smaller share sizes, e.g., 16 bits.

In Example 1, the adversary (thief) is *underinformed*, i.e., lacks the shares needed to recover $\kappa$. Another facet of our research aims to create situations in which an adversary is *overinformed*, having too many shares to identify and extract tag keys. In Appendix A, we consider situations in which an adversary is overinformed when scanning retail shelves where the contents and thus RFID tags of many cases are mixed together.

**Secret sharing across time:** Suppose that $\kappa$ is not an encryption key, but a write-access key. In that case, the ability to recover $\kappa$ by scanning a case would enable a malefactor with access to a single case at any point in the supply chain to modify the data contents of tags. Similarly, suppose that $\kappa$ were a symmetric key used to authenticate tags. Then simply by scanning a case, an adversary could recover all of the key material required to clone the associated tags.

For this reason, we consider another form of secret sharing in which a secret key $\kappa$ is distributed not across the tags in a single case, but across multiple cases. Given that cases—much like data packets—depart and arrive at staggered times in a supply chain, we refer to this approach as secret sharing across time.

**Example 2** *Alice, a manufacturer, is shipping cases of RFID-tagged items to Bob. She would like to communicate the write-access PINs for the tags in these cases to Bob as securely as possible.*

*Suppose that Alice employs trucks that hold up to ten cases. She might do as follows. She selects a window, i.e., sequence, of eleven cases* $c_j, c_{j+1}, \ldots, c_{j+10}$ *designated for delivery to Bob. She creates a master secret* $\kappa$ *from which it is possible to derive the write-access PIN for any tag within the window of cases. She distributes* $\kappa$ *into eleven shares* $S_1, S_2, \ldots, S_{11}$ *via an (11,11)-secret sharing scheme, and writes share* $S_d$ *to case* $c_{j+d-1}$*. (She might distribute the secret across tags on individual items, or on a case-specific tag.)*

*An adversary that gains access to the contents of a small collection of cases, or even an entire truckload, is unable to reconstruct the secret* $\kappa$ *or to obtain the write-access PINs for the RFID tags. On the other hand, Bob can reconstruct* $\kappa$ *once he receives the full sequence of eleven constituent cases.*

Of course, in practice it may be difficult for Alice to identify *a priori* a window of cases that a legitimate receiver, Bob, will receive in its entirety, particularly if the cases pass through intermediaries. Hence the main thrust of our work here is the development of more flexible secret sharing schemes. We propose what we call *Sliding-Window Information Secret-Sharing* (SWISS) schemes, constructions such that for a sequence $c_1, c_2, \ldots$ of cases, Bob need only receive a minimal number $k$ of cases in any contiguous window of size $n$ in order to reconstruct the associated secret keys. SWISS schemes provide key confidentiality against adversaries that intercept cases on a sporadic basis.

As we explain, it is a straightforward matter to create a SWISS scheme in which shares are linear in $n$, and thus potentially large in practice. Our contribution is a SWISS scheme whose shares are constant in size, i.e., have length independent of $k$ and $n$.

## 4 Related Work

Since its invention in 1979 by Shamir [32] and independently by Blakley [4], secret sharing has played a foundational role in cryptography. However, our work differs from previous work in two key aspects: the privacy goal we adopt and the size of the shares employed.

The majority of secret sharing literature evaluates the privacy of a secret-sharing scheme from an information-theoretic perspective, seeking to create efficient schemes for various access structures. In this regime, a perfect secret-sharing (PSS) scheme is one in which an adversary learns no information about the secret in an information-theoretic sense (i.e., even if the adversary has unbounded computational resources). Shamir's scheme [32] qualifies as a PSS scheme. Statistical secret-sharing (SSS) schemes, such as Blakley's [4], allow a small amount of information leakage, in the information-theoretic sense.

A narrower literature concerns complexity (or computational) theoretic secret-sharing (CSS), in which privacy depends on computational bounds on an adversary. Krawczyk first introduced the notion of a CSS scheme [20], and Bellare and Rogaway later refined and formalized it [2]. Work in this area has focused on privacy based on *all-or-nothing* indistinguishability. In other words, in Krawczyk's construction, an adversary either has no information about the secret or she has complete information about it. In this work, we introduce constructions that accommodate *gradated* key information. This allows us to consider schemes in which the leakage of secret information is proportional to and thus grows *gradually* with the number of revealed shares.

The other dimension in which this work differs from previous work is the length of the shares involved. It is well known that in any natural PSS scheme, the size of every participant's share must be at least that of the secret itself [10, 18]. For specific access structures, stronger lower-bounds have been shown [9].

Any scheme in which shares are shorter than the secret is necessarily imperfect. Ogata and Kurosawa [26] give information-theoretic lower bounds on share sizes in such schemes. At a high level, they show that a share must have length equal to at least that of the "gap" in knowledge between sets of shares outside the permitted access structures and the secret itself. More formally, suppose that a secret $x \overset{R}{\leftarrow} D$ is selected at random from distribution $D$. Let $\hat{x}$ denote a random variable for $x$ and $\hat{S}_i$ one for $S_i$, i.e., the $i^{th}$ share generated by a natural secret-sharing scheme. If $\Gamma$ represents the set of access structures that are allowed to recover the secret, then it is the case that $H(\hat{S}_i) \geq min_{\gamma \notin \Gamma} H(\hat{x} \,|\, \{\hat{S}_i\}_{i \in \gamma})$, where $H(A\,|\,B)$ denotes the entropy of $A$ conditional on $B$.

In terms of concrete proposals, in the information-theoretic literature, McEliece and Sarwate note that Shamir's scheme can be generalized as a Reed-Solomon code, permitting a tradeoff between share size and security [25]. Blakley and Meadows propose a class of ramp secret sharing schemes [5] which define two thresholds. Given $t$ shares, it is easy to reconstruct the secret. Less than $t'$ shares reveals no information about the secret, and given some number of shares $y$ such that $t' \leq y < t$, the information gained about the secret is proportional to $\frac{y-t'}{t-t'}$. Larger "ramps" provide weaker security but allow a reduction in share size. In both of these proposals, the size of the shares is dependent on the size of the secret.

By moving to the CSS realm, Krawczyk introduces a scheme with "short" shares with lengths independent of the secret's size [20]. A cryptographic key is shared using a PSS scheme, while the secret is encrypted using the key. The resulting ciphertext is shared using an information-dispersal algorithm, e.g., Rabin's IDA [27]. A share then consists of a cryptographic portion and a ciphertext portion. The cryptographic portion is at least as long as a cryptographic secret key plus a hash function image (thus, in practice, at least 384 bits). We use a similar mechanism to make the size of our shares independent of the secret, but in lieu of PSS and IDA schemes, we employ error correcting codes to reduce share sizes and add robustness.

We are aware of no investigation, however, of the particular problem of creating shares smaller than the short ones introduced by Krawczyk, i.e,, shares potentially *shorter* than a cryptographic secret key (perhaps 16 bits in length). Here, we characterize such shares as *tiny*.

The omission from the literature of CSS schemes with tiny shares appears to have two underlying causes. First, short shares are compact enough for many applications. Second, the literature is solidly anchored in PSS. Even CSS schemes, such as that of Krawczyk, typically rely on PSS as a primitive to share out cryptographic keys.

**Secret-sharing in RFID:** Langheinrich and Marti suggest using secret sharing to conceal an RFID tag's information from adversaries with time-limited access to the tag [21]. The tag's information is split using Shamir's scheme [32], and the tag periodically emits a share. A reader that probes the tag over the course of several minutes will receive enough shares to reconstruct the tag's information, while a casual attacker who only obtains a few emissions cannot reconstruct any tag information. Our schemes, in contrast, spread shares across multiple tags and consider sliding time windows with evolving secrets, rather than a single fixed secret.

In other work, Langheinrich and Marti propose using Shamir's scheme to distribute an item's ID over hundreds of RFID tags integrated into the item's material [22]. They aim to enforce privacy by requiring a reader to access multiple tags. In contrast, we look to dispersion, rather than aggregation, of tags, as a privacy-enforcing mechanism. We also reduce the size of each share to well below the size of standard Shamir shares.

## 5 Secret Sharing Across Space

Sharing a secret (e.g., a cryptographic key) across space in an RFID application imposes severe limitations on the size of each share. As discussed in Section 4, previous schemes typically require 128 bits or more for each share, whereas with RFID tags, we would like shares of 16 bits or less. Hence in this section we provide a generic robust secret sharing scheme that we refer to as a Tiny Secret Sharing (TSS) scheme. We define our scheme in a general problem framework based on adversarial games, describe a prototype implementation, and suggest parameters appropriate for real-world deployment.

### 5.1 Preliminaries

**Secret Sharing.** We adhere closely to the notation and definitions of Bellare and Rogaway [2]. An $n$-party *secret-sharing scheme* is a pair of algorithms $\Pi = (\mathsf{Share}, \mathsf{Recover})$ that operates over a message space $\mathbb{X}$, where:

- $\mathsf{Share}$ is a probabilistic algorithm that takes input $x \in \mathbb{X}$ and outputs the $n$-vector $S \overset{R}{\leftarrow} \mathsf{Share}(x)$, where $S_i \in \{0,1\}^*$. On invalid input $\hat{x} \notin \mathbb{X}$, $\mathsf{Share}$ outputs an $n$-vector of the special ("undefined") symbol $\perp$.
- $\mathsf{Recover}$ is a deterministic algorithm that takes input $S \in (\{0,1\}^* \bigcup \diamondsuit)^n$, where $\diamondsuit$ represents a share that has been erased (or is otherwise unavailable). The output $\mathsf{Recover}(S) \in \mathbb{X} \bigcup \perp$, where $\perp$ is a distinguished value indicating a recovery failure.

In our security definitions, we assume an honest dealer, i.e., correct execution of $\mathsf{Share}$ (although the adversary may choose the secret that is shared).

**Adversaries.** While secret sharing literature traditionally defines goals with respect to access structures, we predicate our definitions below on a class $\mathcal{A}$ of probabilistic adversarial algorithms. We define the security of a TSS scheme in terms of a particular class $\mathcal{A}$. We can reconcile our adversarial model with the traditional access-structure view by restricting $\mathcal{A}$ to only adversaries $A$ that respect a particular access structure. For example, we might consider only adversaries that compromise fewer than $d$ legitimate shares for some $d$.

**Error Correcting Codes.** Our construction utilizes an error-correcting code (ECC), a generalization of secret sharing that we formally define as a pair of algorithms $\Pi^{ecc} = (\mathsf{Share}^{ecc}, \mathsf{Recover}^{ecc})$. An $(N,K,D)_Q$-ECC operates over an alphabet $\Sigma$ of size $|\Sigma| = Q$. $\mathsf{Share}^{ecc}$ maps $\Sigma^K \to \Sigma^N$ such that the minimum Hamming distance in symbols between (valid) output vectors is $D$. For such a function $\mathsf{Share}^{ecc}$, there is a corresponding function $\mathsf{Recover}^{ecc}$ that recovers a message successfully given an attacker that can corrupt up to $D/2$ players or erase the shares of $D-1$ players—or some combination of the two, depending on the specific ECC. (In some cases, correction beyond the minimum distance is possible [28].)

### 5.2 Problem Definition

Informally, the adversary may attack either the privacy or the robustness of the scheme or both. A privacy attacker attempts to recover the secret $x$ given some number of shares. To break robustness, the adversary aims to corrupt shares such that $\mathsf{Recover}$ fails to output $x$. We define these security goals formally below and conclude with a definition of a TSS scheme.

#### 5.2.1 Privacy

We consider two subtypes of privacy attackers: an *underinformed* adversary and an *overinformed* adversary. An underinformed adversary can corrupt a limited number of players, while an overinformed adversary can obtain all $n$ shares, but also receives a number of additional "shares" that she cannot distinguish from the correct shares. Due to lack of space, we relegate details on overinformed adversaries to Appendix A. (Briefly, an overinformed adversary sees shares from multiple cases simultaneously, and cannot feasibly extract secrets due to the hardness of decoding given many "chaff" shares.)

**Underinformed Attacks.** Here, we consider an attacker who obtains a limited number of legitimate shares (recall Example 1). In this setting, Bellare and Rogaway define privacy based on a notion of indistinguishability. Given an $n$-party secret-sharing scheme $(\Pi, \mathbb{X})$, they define the oracle $\mathsf{corrupt}(S,i)$ as a function that returns $S_i$. ("Corruption" in this setting—corresponding to compromise of a share-holding player—results in disclosure, not change, of a share.) Then the Bellare and Rogaway notion of privacy is defined based on the experiment shown in Figure 2(a)

In the experiment, the adversary is asked to choose two values to be shared. The experiment selects one of the secrets at random and generates a set of shares. The adversary can then corrupt (or see the value of) individual shares and must eventually produce a guess as to which secret was shared. The corruptions and the guess may be based on state generated during the "choose" phase. Using this experiment, Bellare and Rogaway define $A$'s advantage as $\mathbf{Adv}_A^{ind}[\Pi,\mathbb{X}] \overset{\triangle}{=} 2\Pr\left[\mathbf{Exp}_A^{ind}[\Pi,\mathbb{X}] \Rightarrow 1\right] - 1$.

#### 5.2.2 Robustness

We desire our scheme to allow a legitimate user to recover the original secret, even if the adversary tampers with some of the shares. To model a scheme's resilience to such an attack, we define a robustness experiment. In

```
Experiment $\mathbf{Exp}_A^{ind}[\Pi, \mathbb{X}]$
    $(x_0, x_1) \leftarrow A(\text{"choose"})$;
    $b \overset{R}{\leftarrow} \{0,1\}$; $S \overset{R}{\leftarrow} \mathsf{Share}(x_b)$;
    $b' \leftarrow A^{\mathsf{corrupt}(S,\cdot)}(\text{"corrupt"})$;
    output '1' if $b = b'$, else '0'
```

(a) **Privacy Experiment**

```
Experiment $\mathbf{Exp}_A^{rec}[\Pi, \mathbb{X}]$
    $x \leftarrow A(\text{"choose"})$;
    $S \overset{R}{\leftarrow} \mathsf{Share}(x)$;
    $S' \leftarrow A^{\mathsf{corrupt}(S,\cdot)}(\text{"corrupt"})$;
    $x' \leftarrow \mathsf{Recover}(\{S'_i\}_{i \in \hat{S}} \bigcup \{S_i\}_{i \notin \hat{S}})$;
    output '1' if $x \neq x'$, else '0'
```

(b) **Robustness Experiment**

Figure 2: **TSS Experiments.** *These experiments capture our notion of privacy and robustness for TSS schemes.*

our robustness experiment, Share is invoked on a secret $x$ of the adversary's choosing. The adversary then corrupts a number of players and *replaces their share values*. Again, the adversary is allowed to maintain state between the "choose" and the "corrupt" phases. The adversary is successful if Recover fails to recover $x$ given the corrupted and uncorrupted shares as input. This experiment is much like that for robustness in Bellare and Rogaway, though their definition additionally includes the technical requirement that the adversary identify an uncorrupted player $j$. This is not necessary for our purposes. We define the robustness experiment as shown in Figure 2(b), letting $\hat{S}$ represent the indices of the shares corrupted by the adversary. We define the advantage of $A$ as $\mathbf{Adv}_A^{rec}[\Pi, \mathbb{X}] \overset{\triangle}{=} \Pr[\mathbf{Exp}_A^{rec}[\Pi, \mathbb{X}] \Rightarrow 1]$.

It is also useful to consider a modified experiment $\mathbf{Exp}^{rec-or-detect}$ that outputs '1' if $x \neq x'$ and $x' \neq \perp$, else '0.' In other words, $A$ is successful if it causes a recovery failure that Recover *does not detect*. This is a weaker requirement, of course, than that represented by $\mathbf{Exp}^{rec}$, but an important condition not explored by Bellare and Rogaway. Given the above experiments, we define a TSS scheme as follows.

#### 5.2.3 TSS Definition

**Definition 1** *A $(k,n)$-TSS scheme is a pair $(\Pi, \mathbb{X})$, such that $\Pi$ distributes n shares of a secret $x \in \mathbb{X}$, of which any set of k correct shares suffices to recover x. The security of the scheme is characterized by an adversary class $\mathcal{A}$ and the tuple: $(q_u, \varepsilon_u, q_r, \varepsilon_r)$, where an underinformed attacker $A \in \mathcal{A}$ making $q_u$ corrupt queries has $\mathbf{Adv}_A^{ind}[\Pi, \mathbb{X}] \leq \varepsilon_u$; likewise, the pair $(q_r, \varepsilon_r)$ applies to robustness attackers. (An extended definition can include overinformed attackers as well; see Appendix A.)*

### 5.3 Our Construction

Figure 3 illustrates a high-level schematic of our TSS scheme. The $\mathsf{Share}^{TSS}$ algorithm accepts as input an arbitrarily-sized secret $x$. It then generates a large random pre-key $\tilde{\kappa}$. We apply a hash to reduce $\tilde{\kappa}$ to the size of a cryptographic key $\kappa$. The hash function also con-

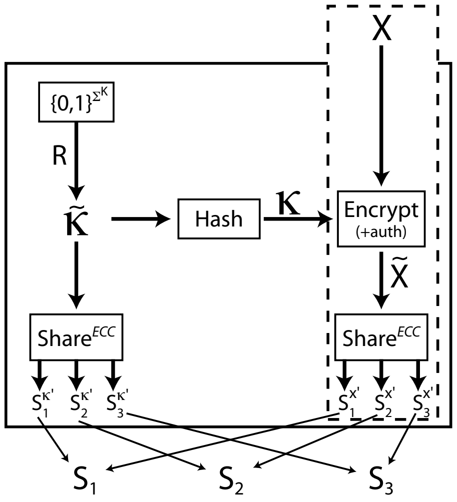


Figure 3: **Secret Sharing with Tiny Shares.** *Schematic of our TSS construction in a toy example with n=3. It can be used to distribute a key $\kappa$, or optionally a secret x of arbitrary size. When $\kappa$ and x are provided at the same time, the two error-correcting codes may be coalesced into a single one.*

stitutes good cryptographic hygiene (and is used in our proofs) in the sense that it renders $\kappa$ indistinguishable even in the face of partial compromise of $\tilde{\kappa}$. We use the key $\kappa$ to perform authenticated encryption of $x$ and then use an $(N,K,D)$-error correcting code (ECC) to share both $\tilde{\kappa}$ and the ciphertext $\tilde{x}$. We focus in this paper on the basic construction that assigns a single symbol to each share. Thus we assume $K = k$. More general constructions are possible, but omitted from this paper. A recipient with enough shares can apply the ECC decoding algorithm to recover $\tilde{\kappa}$ and the ciphertext $\tilde{x}$, and then use $\tilde{\kappa}$ to derive the key $\kappa$ necessary to authenticate and decrypt $x$. In some applications (e.g., transporting the master key used to derive RFID kill codes), we may only want to distribute a key. In that case, we can use $\kappa$ as the desired key, and eliminate the portion of the schematic shown in the dashed box.

Our construction assumes that the hash function behaves as a random oracle [1], and for large secrets, we assume the use of an authenticated encryption mode, such as OCB [29].

Below, we state our claims for the security of this construction. We defer the proofs to Appendix B.

**Claim 1** *Given our construction above, an underinformed attacker's advantage is bounded by* $\varepsilon_u$ *such that*

$$\mathbf{Adv}_A^{ind}[\Pi, \mathbb{X}] \leq \varepsilon_u \leq 1/Q^{k-q_u}.$$

**Claim 2** *Against an attacker who makes* $q_r$ *corrupt queries, if* $q_r < D/2$*, i.e.,* $q_r \leq \lfloor (D-1)/2 \rfloor$*, then* $\mathbf{Adv}_A^{rec}[\Pi, \mathbb{X}] = 0 = \varepsilon_r$*, and if* $q_r \leq D-1$*, then* $\mathbf{Adv}_A^{rec-or-detect}[\Pi, \mathbb{X}] = 0$.

Thus, our construction is a (k,n)-TSS scheme with security tuple $(q_u, 1/Q^{k-q_u}, \lfloor (D-1)/2 \rfloor, 0)$.

**Remark 1** *With an appropriate choice of an ECC, we can generalize the construction above. For example, using a systematic version of Reed-Solomon as the ECC,* $\tilde{\kappa}$ *will be encoded in the initial code symbols. We then apply a hash function (SHA-256 with truncation) to those code symbols to derive* $\kappa$*. If we choose* $Q = 2^{|\tilde{\kappa}|}$ *(and do not release* $S_1^{\tilde{\kappa}}$*), then Share*$^{ECC}$ *becomes a robust PSS scheme, as in Krawczyk's scheme [20]. If we choose* $Q = 2^n$*, then we have the scheme described above. Intermediate choices of* $Q$ *trade increased share size for increased security.*

## 5.4 Implementation Sketch and Real World Parameterization

We implemented a $(15,20)$-TSS scheme using a ThingMagic Mercury5 reader and commercially-available Alien Squiggle Gen2 tags. A schematic view of the setup is shown in Figure 4. Use of a $(15,20)$-TSS scheme means that of the 20 available tags, we need to read at least 15 tags successfully to recover the key and decrypt tag data. We work over the field $GF(2^{16})$, so a share (codeword symbol) is 16 bits. The Share algorithm was then implemented as follows. We chose a secret key $\kappa$ of length 128-bits; we obtained $\kappa$ by choosing a random 240-bit value $\tilde{\kappa}$, hashing it with SHA-256, and then taking the first half of the output. We then encoded $\tilde{\kappa}$ into 20 16-bit symbols with a $(20,15)$ Reed-Solomon ECC using the built-in Reed-Solomon encoder in Matlab's Communication Toolbox. This resulted in 20 16-bit shares, one for each tag.

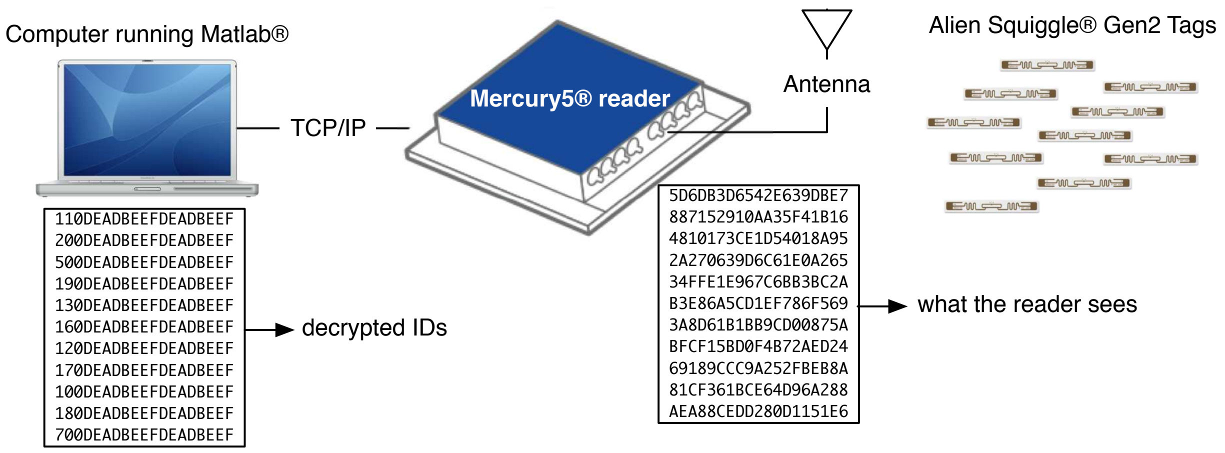
Given that we were using 96-bit tags, we had 80 bits left over for the tag ID. This particular parametrization requires a cipher with an 80-bit block size. We achieve this by using the Blowfish block cipher [31], which has a block size of 64 bits, with Ciphertext-Stealing [11] to expand the block size to 80 bits. Integrity protection at the individual tag ID level is provided by the Gen2 protocol.

Each tag ID $m_i, 1 \leq i \leq 20$, was then replaced by $E_\kappa[m_i]$ and concatenated with a share of $\tilde{\kappa}$ (as generated above). This combined 96-bit string was written into the tag using the same setup (Figure 4). Because all Gen2 RFID readers can also wirelessly write to tags, this operation is accomplished by bring each tag into the antenna field of the reader and executing a Gen2 write command. In practice, this operation would be carried out when the case, pallet, or item tag is initially encoded in the supply chain. Note that $E_\kappa$ as used here includes Ciphertext-Stealing as described above.

For the Recover algorithm, we simply unwound Share. As shown in Figure 4, the reader sees encrypted tag IDs with concatenated shares. As long as the reader sees more than 15 tags, Recover running on the PC outputs the tag IDs successfully.

In an ECC, a codeword consists of an *ordered* sequence of symbols. Because there is no fixed reading order for tags in our implementation, however, it must be *order invariant*. That is, since shares are not distributed among players with fixed identities, as in our robustness experiment, we must explicitly associate an index with each share (effectively assigning a player index to each tag). Thus, the symbol on a tag must be accompanied by an index specifying its codeword position. Rather than specifying this index explicitly, and thereby using an additional 16 bits of storage, we derive it implicitly based on the encrypted tag ID. In particular, we hash the ID using SHA-256, and interpret the last 16 bits as the index; of course, we must do this *before* sharing the encryption key. This optimization potentially introduces a new problem: Two (or more) tags within a case may have ciphertexts that hash to the same index. A sufficiently large index size can minimize this problem. (By the Birthday Paradox, $GF(2^{16})$ accommodates roughly 256 tags without many collisions.) As a further optimization, we can dedicate a few additional bits of storage to disambiguating collisions that do occur. Finally, if there are still too many collisions, we can simply choose a new random pre-key $\tilde{\kappa}$ and compute a new set of shares.

In general, the first step in parameterizing the TSS scheme for real-world usage is to determine the total number of tags $n$ and the key-recovery threshold $k$. As noted earlier (section 2.2), these numbers can vary widely between use cases. Today, pallets typically carry from 1 to 200 tags each. In a typical distribution center setting, an RFID reader could, depending on pallet composition, fail to read as many as 2–3% (i.e., 4–6) of the tags in a 200-item pallet, and it may pick up as many as 3–10 *stray* tags from a pallet in an adjacent dock door. This means that we can see up to 6 erasures, and up to 10 errors in reading. These numbers are borne out by one the authors' (RP) long experience in supply chain RFID deployments. Thus the choice of a (200, 170)-Reed Solomon code (the minimum distance $D = N - K + 1$ is typically omitted from Reed-Solomon parameterization), which can correct up to 15 errors or 30 erasures, would provide sufficient error correction for real-world deployments. As discussed in Section 2.2, individual consumers typically have fewer than 40 tags from the same case, so we could alternatively choose a (200, 40)-Reed Solomon code to maintain privacy and provide additional robustness to read errors.



```
110DEADBEEFDEADBEEF
200DEADBEEFDEADBEEF
500DEADBEEFDEADBEEF
190DEADBEEFDEADBEEF
130DEADBEEFDEADBEEF
160DEADBEEFDEADBEEF
120DEADBEEFDEADBEEF
170DEADBEEFDEADBEEF
100DEADBEEFDEADBEEF
180DEADBEEFDEADBEEF
700DEADBEEFDEADBEEF
```

```
5D6DB3D6542E639DBE7
887152910AA35F41B16
4810173CE1D54018A95
2A270639D6C61E0A265
34FFE1E967C6BB3BC2A
B3E86A5CD1EF786F569
3A8D61B1BB9CD00875A
BFCF15BD0F4B72AED24
69189CCC9A252FBEB8A
81CF361BCE64D96A288
AEA88CEDD280D1151E6
```

Figure 4: **Schematic of implementation setup** *20 TSS-encoded RFID tags, at far right, are prepared using* Share *as described in the text. They are read by a ThingMagic Mercury5 reader and the encrypted IDs are passed over the network to a Matlab program running* Recover *on a computer. The computer first recovers the Reed-Solomon-encoded secret key and then decrypts the tags. The two boxes below the schematic depict what the reader sees and the eventual decrypted tag IDs. In practice,* Recover *would be ported to run directly on the reader. Given the capabilities of current RFID readers, direct implementation on the reader is straightforward.*

Lastly, we remark on the choice of the field size. As the field size is the main determinant of the extra tag memory consumed by our scheme, smaller fields mean smaller memory requirements. Larger field sizes reduce the number of index collisions, which is useful both to ensure good decoding rates and to enforce security against an overinformed adversary (Appendix A). In applications where only the underinformed attacker must be considered, we can potentially reduce the space on each tag to a single bit, for sufficiently large $k$ and an appropriate ECC scheme.

## 6 Secret Sharing Across Time

Thus far, we have considered sharing schemes for one shipment. However, a distributor may wish to increase security by leveraging the fact that a legitimate recipient should receive more shipments than an attacker can access (recall Example 2 from Section 3). In this section, we explore a class of schemes that uses such information disparities across sliding time windows. In the future, we will investigate schemes leveraging the entropy of the entire history of interactions between a sender and recipient.

### 6.1 Defining SWISS: Sliding-Window Information Secret Sharing

In the schemes below, we assume a sender periodically emits a share $S_i$. For RFID purposes, we may suppose the sender is a manufacturer who periodically ships out containers of RFID-labeled items. Each share may optionally be further shared out amongst the RFID tags in the container as described in Section 5. Each period also has an associated key $\kappa_i$. Thus, we have a sequence of shares $S = \{S_0, S_1, \dots\}$ that expands indefinitely over time. We

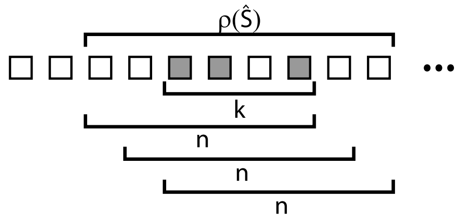


Figure 5: *In this example, if the adversary holds a set $\hat{S}$ of $k = 3$ shares (shown as shaded boxes), then we define $\rho(\hat{S})$ as the union of all (three) windows of $n = 6$ shares containing the original $k$ shares. We require that the adversary be unable to recover keys for periods outside of $\rho(\hat{S})$. The figure assumes $\lambda = 0$. If $\lambda = 1$, then $\rho(\hat{S})$ would include two additional shares: one before and one after the set $\rho(\hat{S})$ currently shown.*

assume that within any window of $n$ elements, only a legitimate recipient receives at least $k$ of the shares in that window, and given those shares, the recipient should be able to recover the corresponding keys. An adversary receiving fewer shares should learn nothing about the keys.

More formally, a SWISS scheme is defined as a pair of algorithms $\Pi = (\mathsf{Share}, \mathsf{Recover})$, where:

- $\mathsf{Share}(k, n, \tau)$ is a probabilistic algorithm that takes as input a threshold for recoverability $k$, a window size $n$, and a security parameter $\tau$. It outputs two "infinite" vectors $\kappa$ and $S$, where $\kappa_i \in \{0,1\}^\tau$ is the key for period $i$, and $S_i$ is the share for period $i$. On invalid input, Share outputs the special symbol $\perp$.
- Recover is a deterministic algorithm that takes as input $S' \subset W_j$ where $W_j$ defines a sequence of $n$ shares starting at time $j$ such that $W_j = \{S_i : j \le i < j + n\}$, and $|S'| \ge k$. The output of $\mathsf{Recover}(S')$ is a set of keys $K = \{\kappa_i : S_i \in S'\}$ for the shares provided in $S'$ or $\perp$, a special value indicating a recovery failure.

In our security definitions, we again assume an honest dealer, i.e., correct execution of Share. Below, we give formal definitions for our privacy and recoverability requirements.

**Privacy.** To define privacy, we require that the adversary cannot obtain the key for any share she does not possess. If the adversary holds fewer than $k$ shares, she should not learn any keys. We deal with the case in which the adversary holds more than $k$ shares as follows.

Define the set of shares held by the adversary as $\hat{S}$. Let $\rho(\hat{S})$ be the set of all shares that lie in a window of size $n+\lambda$ for which the adversary has recovered at least $k$ shares. We require the adversary to be unable to recover the key for any element in $\overline{\rho}(\hat{S})$, the complement of $\rho(\hat{S})$. Since $k$ shares allow the adversary to recover all of the keys in a window of size $n$, the value of $\lambda$ indicates the amount of information $k$ shares "leak" about keys not contained within a window of $n$ shares. Figure 5 illustrates these requirements.

More formally, we can define privacy based on the following experiment:

```
Experiment $\mathbf{Exp}_A^{ind-swiss}[\Pi]$
    $(S,\kappa) \overset{R}{\leftarrow} \mathsf{Share}(k,n,\tau)$;
    $i \leftarrow A(\text{"choose"})$;
    $\kappa^R \overset{R}{\leftarrow} \{0,1\}^{|\kappa|}$; $b \overset{R}{\leftarrow} \{0,1\}$;
    $b' \leftarrow A^{\mathsf{corrupt}(S,\cdot)}(\pi(b,\kappa^R,\kappa_i),\text{"corrupt"})$;
    if $i \notin \rho(\hat{S})$ or $i \notin \hat{S}$ then
        output '1' if $b' = b$, else '0';
    else output '0';
```

where $\pi(0,x,y) = (x,y)$ and $\pi(1,x,y) = (y,x)$. Essentially, the adversary is asked to choose a time period $i$. After corrupting some number of shares, the adversary must distinguish between the key for period $i$ and a randomly selected key. We consider the adversary successful if the period chosen does not correspond to a share held by the adversary, or if the period lies outside the set $\rho(\hat{S})$ induced by the adversary's selection of shares. The adversary's advantage is then $\mathbf{Adv}_A^{ind-swiss}[\Pi] \overset{\triangle}{=} 2\mathsf{Pr}\left[\mathbf{Exp}_A^{ind-swiss}[\Pi] \Rightarrow 1\right] - 1$.

**Recoverability.** We require that any set $S' \subseteq W_j$ with $|S'| \geq k$ shares suffices to recover the keys associated with each share in the set, namely $\{\kappa_i : S_i \in S'\}$. We define recoverability for a legitimate recipient in the erasure model; in other words, shares may be lost but not corrupted. We can convert our SWISS schemes to a corruption model by replacing our use of PSS schemes with robust PSS schemes, such as Krawczyk's [20].

**Definition 2** *We define a $(k,n)$-SWISS scheme as a pair of algorithms $\Pi$ as defined above where* Share *produces shares of size $\mu$. The security is characterized by the pair $(\lambda,\varepsilon)$, where (as explained above) $k$ shares are sufficient to reveal $\lambda$ "nearby" keys for time periods not contained in a window of $n$ shares, and $\mathbf{Adv}_A^{ind-swiss}[\Pi] \leq \varepsilon$.*

Thus, an ideal SWISS scheme would have $(\lambda,\varepsilon) = (0,0)$ with minimal $\mu$.

## 6.2 A Family of SWISS Schemes

In our SWISS construction, we want to ensure that the secret for a case is only available given possession of that case. To achieve this property, we make the key $\kappa_i$ for case $i$ a function of both a window key and a secret value associated with the case (or its RFID tag).

Ideally, the window key for a window of $n$ cases should be recoverable if and only if the receiver possesses $k$ or more cases within that window. A naïve SWISS scheme would simply generate a key for every possible window of size $n$ and share each key using a $(k,n)$ scheme. But a case would then need a share for every window covering it, and hence the per-case share size would grow linearly with the size $(n)$ of each window.

Instead, we aim to bring the share size down to a small constant independent of $k$ and $n$. We use two techniques for this goal. First, we allow some sloppiness in our access structure. Our access structure (in our main construction) depends on superwindows of size $2n$ that each overlap with the previous superwindow by $n$ (see Figure 6); each superwindow secret is shared using a $(k,2n)$ scheme. Access to a window secret requires recovery of the secrets for either one of its two corresponding superwindows. Any $k$ shares in a sequence of size $n$ fall into some superwindow of size $2n$, and therefore allow recovery of the superwindow secret. The "sloppiness" is this: Recovery of shares in one window allows for recovery of secrets in nearby windows.

Given the superwindow scheme described above, we could encrypt the secret $\kappa_i$ for each case $i$ under each of its corresponding superwindow secrets, $\sigma$ and $\sigma'$. However, using a second technique based on bilinear maps, we can derive a common secret directly from either of the two superwindow secrets $\sigma$ *or* $\sigma'$.

Below, we first explain the assumptions necessary for our schemes. Then we present our main SWISS construction (Section 6.2.2) and show how to generalize it to a wider range of parameters (Section 6.2.3).

### 6.2.1 Assumptions

Our family of SWISS schemes uses bilinear pairing to reduce storage costs. In the full version of this paper, we describe a variant of our SWISS construction based on the more standard RSA assumption. Unfortunately, that version does not generalize efficiently to large window sizes in the same way as does the bilinear map scheme, and hence we focus on the latter.

We give some very brief preliminaries on bilinear maps, referring the reader to [7] for details. Let $E$ be a multiplicative cyclic group of prime order $p$ under a bilinear operator $\hat{e}$ as defined in Boneh-Franklin [7]. Thus we have $\hat{e}: E \times E \to E'$ for a second group $E'$. The bilinear operator $\hat{e}$ has the property that $\forall G, H \in E, \hat{e}(G^a, H^b) = \hat{e}(G,H)^{ab}$; it is also non-degenerate, meaning that if $G$ is a generator of $E$, then $\hat{e}(G,G) \neq 1$.

Our work relies on the hardness of a slightly modified Bilinear Diffie-Hellman Exponent (BDHE) problem [6,8]. Specifically, let $g$ and $\gamma$ be random generators of $E$, and $\alpha$ be a random element in $\mathbb{Z}_p^*$. Our $(\ell, L)$-BDHE problem is defined as:

$$\begin{aligned} &\text{Given } g, \gamma, g^{(\alpha^i)} && \text{for } i = 1,2,\ldots,\ell-L, \ell+1, \ldots, 2\ell \\ &\qquad \text{and } \gamma^{(\alpha^i)} && \text{for } i = 1,2,\ldots,L-1 \\ &\text{compute } \hat{e}(g,\gamma)^{(\alpha^\ell)}. \end{aligned}$$

In the original framing of the $\ell$-BDHE problem [6, 8], only $\gamma$ (rather than additional $\alpha$ powers of $\gamma$) is assumed to be known. We assume that $L \geq 2$, since the $(\ell,1)$-BDHE problem simply degenerates to the $\ell$-BDHE problem. Loosely speaking, the $(\ell,L)$-BDHE assumption in $E$ says that no efficient algorithm can solve the $(\ell,L)$-BDHE problem in $E$ with non-negligible probability.

We can apply the "master" theorem of Boneh et al. [6] to bound the difficulty of $(\ell,L)$-BDHE in a generic group. In their terminology, we have $P = (1, y, y^2, \ldots, y^{L-1}, x, x^2, \ldots, x^{\ell-L}, x^{\ell+1}, \ldots, x^{2\ell})$, $Q = (1)$ and $f = x^\ell y$. This implies that an attacker $A$ with advantage $1/2$ in solving the decision $(\ell,L)$-BDHE problem in a generic bilinear group $E$ must take time at least $\Omega\left(\sqrt{p/(4\ell)} - 2\ell\right)$. E.g., if we assume the distributor sends one billion windows (or less), then solving the decision $(\ell,L)$-BDHE problem in a generic bilinear group $E$ of size 192 bits takes time at least $2^{80}$. Of course, a lower bound in a generic group does not imply a lower bound in any specific group.

### 6.2.2 Our Main SWISS Construction

In Section 6.2.3, we present a fully generic overlapping SWISS scheme, but first, to simplify the exposition, we describe a single member of the family (see Figure 6). This example provides a $(k,n)$-SWISS scheme with $\mu = 3\tau$ and security parameters $(2n-k, \varepsilon)$.

Starting at time 0, the sender defines a series of superwindows $W_0, W_n, W_{2n}, \ldots, W_{\ell n}$, each of size $2n$. Thus, each superwindow consists of two windows of size $n$, with one window overlapping a window from the previous superwindow, and one window overlapping a window from the subsequent superwindow. Each superwindow $W_{\ell n}$ defines a $(k, 2n)$ perfect secret sharing (PSS) of the superwindow secret $\sigma_{\ell n}$. Since each time period $i$ is covered by two superwindows, each comprising its own secret sharing scheme, the share $S_i$ distributed in each time period

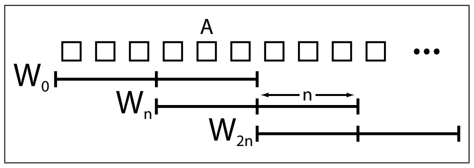


Figure 6: *Each superwindow of $2n$ shares (in the example shown here, $n = 3$) overlaps with the previous superwindow by $n$ shares. Each superwindow $W_{\ell n}$ is a $(k, 2n)$ sharing of the superwindow secret $\sigma_{\ell n}$. Each time period is covered by two superwindows. For example, the share labeled A is covered by superwindows $W_0$ and $W_n$. As a result the key for that period $\kappa_A$ can be recovered from either superwindow secret, $\sigma_0$ or $\sigma_n$.*

consists of two sub-shares $(s_i^{\ell n}, s_i^{(\ell+1)n})$, one for $\sigma_{\ell n}$ and one for $\sigma_{(\ell+1)n}$. We also augment the share with a random nonce $r_i \overset{R}{\leftarrow} \{0,1\}^\tau$. Thus, the share emitted during time period $i$ is $S_i = (s_i^{\ell n}, s_i^{(\ell+1)n}, r_i)$.

Because any time period $i$ is covered by two superwindows (say $W_{\ell n}$ and $W_{(\ell+1)n}$), we would like the key $\kappa_i$ to be recoverable from the superwindow secret of either one (since we do not know a priori in which superwindow the recipient will have $k$ shares). Like many problems in computer science, we can solve this by adding another layer of indirection. Let $y, z \in E$, $a \in \mathbb{Z}_p^*$, and let $(P_0, P_1) = (y, y^a)$ be a public key. Let each of the superwindow secrets be defined so that $\sigma_{\ell n} = z^{a^\ell}$. We define a series of window secrets $\omega_0, \omega_n, \ldots, \omega_{\ell n}$ so that

$$\omega_{\ell n} = \hat{e}(P_1, \sigma_{\ell n}) = \hat{e}(P_0, \sigma_{(\ell+1)n}) = \hat{e}(y,z)^{a^{\ell+1}}.$$

That is, knowing $\sigma_{\ell n}$ allows a recipient to derive $\omega_{\ell n}$ and $\omega_{(\ell+1)n}$.

Finally, we define each key $\kappa_i$ based on the window it belongs to, as well as the random nonce $r_i$ distributed with share $S_i$, as $\kappa_i = h(r_i, \omega_{kn})$, where $h: \{0,1\}^* \to \{0,1\}^\tau$ is a hash function modeled as a random oracle [1].

In the next section, we show how to generalize this construction to decrease $\lambda$ at the cost of increasing the size of each share. We can define an adversary for this more general scheme as follows:

**Definition 3** *We define an $(\ell, L, q)$-adversary $A$ as an attacker who achieves an $\mathbf{Adv}_A^{ind-swiss}[\Pi] < \varepsilon$ advantage in our privacy experiment (defined in Section 6.1), where $\Pi$ is an instantiation of our generic SWISS family with $\Psi = L - 1$ (for $L \geq 2$) that produces at most $2\ell$ shares. The adversary makes at most $q$ random oracle queries.*

In Appendix C, we use this definition to demonstrate the security of the generalized scheme (and hence this specific instantiation) by proving the following theorem:

**Theorem 1** *For any polynomial-time $(\ell, L, q)$-adversary $A$ with $\mathbf{Adv}_A^{ind-swiss} = \varepsilon$ and $\ell > L \geq 2$, there is a polynomial-time adversary $A'$ that solves the $(\ell,L)$-BDHE problem with probability $(\varepsilon - 2^{-\tau})/q\ell - 1/2^\tau$.*

Essentially the theorem states that given an adversary who achieves a non-negligible advantage in our privacy experiment, we can construct an attacker who violates the $(\ell, L)$-BDHE assumption. We also demonstrate that this construction satisfies our recoverability requirement.

**Remark 2** *As described, our SWISS construction uses a PSS scheme to create superwindow shares. Thus, the construction tolerates erasures but not errors. However, we could readily replace the PSS scheme with a robust scheme, such as our TSS scheme from Section 5, which would both decrease the size of the individual shares and add error tolerance to the SWISS construction.*

### 6.2.3 A Generic SWISS Family

The above scheme can be generalized to allow decreased values of $\lambda$ at the cost of increased storage (see Figure 7). Specifically, for any value of $\Psi < n$, we can create a $(k,n)$ SWISS scheme with $\mu = (\Psi+2)\tau$ and security parameters $((1+\frac{1}{\Psi})n-k, \varepsilon)$.

Essentially, we divide each superwindow $W$ into $\Psi+1$ windows of size $\frac{n}{\Psi}$. The superwindows form $(k, \frac{(\Psi+1)n}{\Psi})$ sharing schemes of the superwindow secrets, and each superwindow overlaps the previous superwindow by $\Psi$ windows. Thus, any given window is covered by $\Psi+1$ superwindows, and the window secret can be recovered from any of the superwindow secrets, using the same elliptic curve pairings technique as before. In other words, we define a public key $(P_0, P_1, ..., P_\Psi) = (x, x^a, ..., x^{a^\Psi})$, and a window secret $\omega_{\ell n}$ is defined as:

$$\begin{aligned}\omega_{\ell n} &= \hat{e}(P_\Psi, \sigma_{\ell n}) = \hat{e}(P_{\Psi-1}, \sigma_{(\ell+1)n}) = ... \\ &= \hat{e}(P_0, \sigma_{(\ell+\Psi)n}) = \hat{e}(x,z)^{\ell+\Psi}.\end{aligned}$$

To determine $\lambda$, we consider the worst case, in which $k \leq \frac{n}{\Psi}$, and the adversary's $k$ shares fall within a single window. The window then is covered by $\Psi+1$ superwindows, allowing the adversary to recover secrets for $2\Psi+1$ windows, or $(2\Psi+1)\frac{n}{\Psi} = 2n + \frac{n}{\Psi}$ secrets. These secrets can be at most a superwindow ($\frac{\Psi+1}{\Psi}n$) away from the $k$ secrets held by the adversary, so $\lambda = \frac{\Psi+1}{\Psi}n - k = (1+\frac{1}{\Psi})n-k$. If $k > \frac{n}{\Psi}$, then fewer than $\Psi+1$ superwindows will contain $k$ shares, and hence $\lambda$ will be even smaller.

In our example scheme from Section 6.2.2, $\Psi = 1$, so each superwindow formed a $(k, 2n)$ secret-sharing scheme, but we could also use $\Psi = 2$, with each superwindow consisting of 3 windows of $\frac{n}{2}$ shares, and the superwindow as a whole constituting a $(k, \frac{3}{2}n)$ sharing of the superwindow secret (see Figure 7(a)). This would produce a smaller value of $\lambda = \frac{3}{2}n - k$, but at the cost of larger shares: each issued share would now contain three shares (one for each superwindow overlapping a particular window) and the random nonce $r_i$.

### 6.2.4 Real World Instantiation

To make our SWISS construction more concrete, we suggest sample parameters for real world deployments. Suppose the sender needs to ship one million or fewer shares. We divide those shares into 10,000 windows of 100 shares each, giving us $\ell = 5{,}000, n = 100$. A legitimate recipient will receive at least $k = 20$ shares in any window. If we use the scheme from Section 6.2.2, then $\Psi = 1$ and $L = \Psi + 1 = 2$. Finally, if we use $\tau = 128$ bit keys, then the share for each period will be $3\tau = 384$ bits in size. In contrast, the naïve scheme described earier in this section would require $n\tau = 12{,}800$ bits per share.

We described both our SWISS scheme and the naïve scheme using PSS as a component. If we replace the PSS scheme with our TSS scheme from Section 5, then we have a share size of 16 bits. In our scheme, we still need a random nonce of at least 60 bits, but that yields shares of size $2 \cdot 16 + 60 = 92$ bits, just small enough to fit on an EPC tag. In contrast, the naïve scheme would require $n \cdot 16 = 1{,}600$ bits.

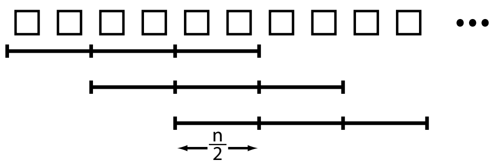
## 7 Conclusions and Future Work

We have described two approaches to secret sharing in unidirectional channels: secret-sharing across space and secret-sharing across time. As we have shown, secret-sharing across space is a tool of practical promise for privacy protection in real-world RFID-enabled supply chains. Our SWISS scheme for secret-sharing across time can, similarly, help address the challenges of RFID tag and reader authentication. An open problem of particular interest in our SWISS construction, however, is elimination of its reliance the non-standard $(\ell, L)$-BDHE problem in our fully generic overlapping SWISS scheme. We also plan to investigate extended SWISS schemes that leverage the entire history of interaction between a sender and receiver, rather than simply a window of recent history.

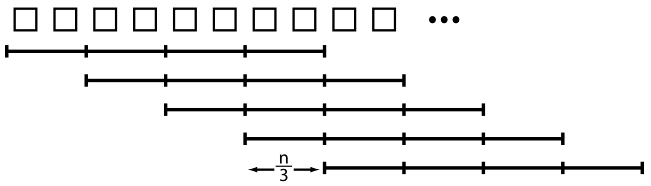
More broadly, we believe that a holistic view of the special operational requirements of RFID tags and the highly constrained resources of tags can give rise to important new cryptographic problems. Our future work will aim to calibrate cryptographic tools like those presented here to RFID supply-chain infrastructure as it evolves and its special operational demands come into clearer focus.

## 8 Acknowledgements

The authors would like to thank Burt Kaliski, Jonathan McCune, Alina Oprea, and Diana Parno for their helpful feedback and editorial suggestions. We are also grateful for the comments from the anonymous reviewers.



(a) **A SWISS scheme with $\Psi = 2, n = 4$.** Each superwindow is a $(k, 3n/2)$ sharing of the superwindow secret.



(b) **A SWISS scheme with $\Psi = 3, n = 6$.** Each superwindow shown is a $(k, 4n/3)$ sharing of the superwindow secret.

Figure 7: **Additional SWISS examples** *We can create additional SWISS schemes by increasing the number of windows per superwindow while decreasing the number of shares in each window. As we increase the number of windows, $\lambda$ decreases, but the number of shares that must be held in each time period increases.*

## References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, pages 62–73, 1993.

[2] M. Bellare and P. Rogaway. Robust computational secret sharing and a unified account of classical secret-sharing goals. In *ACM CCS*, 2007.

[3] C. H. Bennett, G. Brassard, C. Crepeau, and U. Maurer. Generalized privacy amplification. In *ISIT: Proceedings IEEE International Symposium on Information Theory*, 1994.

[4] G. Blakley. Safeguarding cryptographic keys. In *AFIPS Conference Proceedings*, volume 48, pages 313–317, 1979.

[5] G. Blakley and C. Meadows. Security of ramp schemes. In *Advances in Cryptology: Proceedings of CRYPTO*, 1984.

[6] D. Boneh, X. Boyen, and E.-J. Goh. Hierarchical identity based encryption with constant size ciphertext. In *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 440–456, 2005.

[7] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.

[8] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. *Advances in Cryptology: Proceedings of CRYPTO*, 2005.

[9] E. F. Brickell and D. R. Stinson. Some improved bounds on the information rate of perfect secret sharing schemes. *Journal of Cryptology*, 5:153–166, 1992.

[10] R. M. Capocelli, A. D. Santis, L. Gargano, and U. Vaccaro. On the size of shares for secret sharing schemes. *Journal of Cryptology*, 6:157–167, 1993.

[11] J. Daemen. *Hash Function and Cipher Design: Strategies Based on Linear and Differential Cryptanalysis*. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium, Mar. 1995.

[12] EPC Global. EPC® Radio-Frequency Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz – 960 MHz Version 1.1.0. *EPC Global*, 2006.

[13] EPC Global. EPC® Item Level Tagging Joint Requirements Group. *EPC Global*, 2007.

[14] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometry codes. *IEEE Transactions on Information Theory*, 45(6):1757–1767, 1999.

[15] A. Juels. Strengthing EPC tags against cloning. In *ACM Workshop on Wireless Security (WiSe)*, pages 67–76. ACM Press, 2005.

[16] A. Juels, D. Molnar, and D. Wagner. Security issues in e-passports. In *SecureComm*, 2005.

[17] A. Juels and M. Sudan. A fuzzy vault scheme. *Des. Codes Cryptography*, 38(2):237–257, 2006.

[18] E. D. Karnin, J. W. Greene, and M. E. Hellman. On secret sharing systems. *IEEE Transactions on Information Theory*, 29(1):35–41, 1983.

[19] A. Kiayias and M. Yung. Directions in polynomial reconstruction based cryptography. *IEICE Transactions*, E87-A(5):978–985, 2004.

[20] H. Krawczyk. Secret sharing made short. In *Advances in Cryptology: Proceedings of CRYPTO*, pages 136–146, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[21] M. Langheinrich and R. Marti. Practical minimalist cryptography for RFID privacy. In submission, 2007.

[22] M. Langheinrich and R. Marti. RFID privacy using spatially distributed shared secrets. In *Proceedings of UCS 2007*, LNCS, Berlin Heidelberg New York, Nov. 2007. Springer. (To appear).

[23] J. L. Massey. Shift register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.

[24] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-Believing: Using camera phones for human-verifiable authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[25] R. J. McEliece and D. V. Sarwate. On sharing secrets and Reed-Solomon codes. *Communications of the ACM*, 24(9):583–584, 1981.

[26] W. Ogata and K. Kurosawa. Some basic properties of general nonperfect secret sharing schemes. *Journal of Universal Computer Science*, 4(8), 1998.

[27] M. O. Rabin. The information dispersal algorithm and its applications, 1990.

[28] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal SIAM*, 8:300–304, 1960.

[29] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM TISSEC*, Nov. 2001.

[30] N. Sastry, U. Shankar, and D. Wagner. Secure verification of location claims. In *ACM Workshop on Wireless Security (WiSe 2003)*, pages 1–10, Sept. 2003.

[31] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In R. J. Anderson, editor, *FSE*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 1993.

[32] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[33] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Security Protocols, 7th International Workshop*. Springer Verlag, 1999.

## A Overinformed Adversaries

In the body of the paper, we discuss the notion of an underinformed adversary, one that has an insufficient set of shares to reconstruct a secret key. We also briefly consider an *overinformed* adversary., one that possesses a set of shares sufficient to reconstruct one or more secret keys, but has too many shares to feasibly determine such keys. We can design our system such that an adversary is overinformed in settings where the adversary is forced to scan the contents of not one, but *multiple* cases simultaneously.

Consider, for example, an attacker who periodically scans a store shelf, hoping to accumulate enough shares to recover the associated key. The adversary's reader may receive responses from items that arrived in multiple independent cases. In this situation, we would like it to be hard for the adversary to recover any case secret from the full set of secrets, even if a subset of the adversary's shares would suffice to reconstruct the secret. We can appeal to the fact that when shares from multiple cases are mixed together, the large set of shares can make it hard to decode any individual secret.

To help render an attacker overinformed, we can deliberately introduce "chaff" among the shares $S_i$ in a case. Essentially, we replace $\zeta$ shares of $\tilde{\kappa}$ with randomly chosen values. The choice of $0 \leq \zeta < D/2$ represents a tradeoff between security against an overinformed attacker and the error-tolerance of the scheme. For example, by choosing $\zeta = \frac{D}{3}$, an adversary who recovers the shares from two secrets will hold $\frac{2D}{3}$ chaff values—potentially exceeding the recovery threshold for the ECC scheme, as we now show. In this situation, however, a legitimate recipient can still tolerate $\frac{D}{6}$ errors in the shares she receives.

The following experiment formalizes the notion of an overinformed adversary.

```
Experiment $\mathbf{Exp}_A^{ind'}[\Pi, \mathbb{X}, \alpha, \beta]$
    $(x_1, ..., x_\alpha) \overset{R}{\leftarrow} \mathbb{X}$;
    $C \overset{R}{\leftarrow} \bigcup_{i=1}^{z} C^i$, where $C^i \overset{R}{\subseteq} \mathsf{Share}(x_i)$, and $|C^i| = \beta$;
    $H \leftarrow \{h : h = \mathbb{H}(x_i), 1 \leq i < \alpha\}$;
    $x' \leftarrow A^{\mathsf{corrupt}(C,\cdot)}(H, \text{"corrupt"})$;
    output '1' if $x \in (x_1, ..., x_\alpha)$, else '0'
```

In this experiment, we choose $\alpha$ random secrets. The adversary has access to an unlabeled set of shares, which contains $\beta$ randomly chosen shares from each secret. The adversary also receives the hash $\mathbb{H}$ of each secret. Given this information, the adversary must recover one of the original secrets. In this experiment, we define the advantage of adversary $A$ as $\mathbf{Adv}_A^{ind'}[\Pi, \mathbb{X}, \alpha, \beta] \overset{\triangle}{=} \mathsf{Pr}\left[\mathbf{Exp}_A^{ind'}[\Pi, \mathbb{X}, \alpha, \beta] \Rightarrow 1\right]$.

We can characterize the overinformed adversary's task in terms of the *polynomial reconstruction (PR)* problem, the decoding of a Reed-Solomon codeword in the presence of errors (see [19] for detailed discussion).

Given an underlying $(N, K)$-Reed-Solmon code, and a set of $t$ symbols, of which $\zeta$ are corrupted, the classical Peterson-Berlekamp-Massey (PBM) algorithm [23] successfully decodes a set of symbols if $t - \zeta \geq (t + K)/2$ (or, equivalently, $\zeta \leq (t - K)/2$. A more powerful decoding scheme is that of Guruswami and Sudan (GS) [14], which successfully decodes for $t - \zeta > \sqrt{KN}$ in any field of cardinality at most $2^N$. It is conjectured that decoding beyond the error bound represented by GS is infeasible in a general sense and thus that GS offers a likely bound on the hardness of the PR problem.

That said, there are different formulations of the PR problem and little work on the concrete hardness of the problem. Schemes that achieve unconditional security, e.g., [17] do not offer attractive parameterization ranges for our purposes. Choosing credible and practical hardness assumptions for an overinformed adversary in our scheme is an open problem.

## A.1 Parameterization of Our RFID Secret-Sharing Scheme

We give a brief characterization of what we believe to be secure and feasible parameterizations of our scheme. These parameterizations permit PBM decoding for the legitimate reading of a single RFID-tagged case and at the same time exceed the GS bound for security against overinformed adversaries. We emphasize, however, that further research is needed for firm determination of the security of our scheme in a concrete sense.

Suppose that a case contains $N$ tags, of which $\zeta$ are chaff. PBM decoding for a scanned case is always possible when the number of corruptions (or erasures) of valid symbols $e$ is such that $N-(e+\zeta) \geq (N+K)/2$.

**Example 3** *Suppose that $K=8$, $N=200$, and $\zeta=86$. Then it is possible to recover the secret associated with a case for $e \leq 10$, and thus up to a 5% corruption of tag symbols.*

Suppose that an adversary reads symbols associated with $q$ cases and attempts to recover the secret $x$ associated with a particular case. We can establish a lower bound on the hardness of this problem by rendering the problem easier for the adversary. In particular, let us assume that the adversary has access to an oracle that identifies valid shares associated with the $q-1$ untargeted cases (but does not otherwise reveal which shares correspond to which case). Then the adversary can reduce the problem of recovering $x$ to a decoding problem with $N-\zeta$ valid shares and $\zeta q$ chaff shares, and thus $t=N+(q-1)\zeta$ shares in total. The GS bound implies that recovery of $x$ is hard if $N-\zeta < \sqrt{K(N+(q-1)\zeta)}$.

**Example 4** *Suppose that $K=8$, $N=200$, and $\zeta=86$. Then the problem of recovering a target case secret $x$ is hard under the GS bound if $114 < \sqrt{848+688q}$, and thus for $q \geq 18$.*

A stronger bound is possible assuming that valid symbols, i.e., secret-bearing data, in untargeted cases may be regarded as chaff. This gives us a slightly unorthodox problem distribution in which a problem instance has $q$ embedded, secret polynomials. In this case, however, the GS bound implies that recovery of $x$ is hard if $N-\zeta < \sqrt{qKN}$. With an appropriate parameter choice, we can obtain strong concrete results.

**Example 5** *Suppose that $K=100$, $N=200$, and $\zeta=40$ (giving a 5% correction buffer in the single-case setting, as above). Then the problem of recovering a target case secret $x$ is hard under the GS bound if $160 < \sqrt{20000q}$, and thus for $q \geq 2$.*

## B Proofs of Security for Our Tiny Secret Sharing (TSS) Scheme

### B.1 Proof of Privacy

Since many of our applications only require the distribution of a secret key, we first define a simplified experiment to measure the indistinguishability of $\kappa$. Note that for this experiment, we excise the portion of our scheme in the dotted box in Figure 3. Effectively, we share out a null secret $x$, and write $\mathsf{Share}()$ to indicate this fact. The proof of privacy for secrets of arbitrary size then follows in a straightforward manner.

We define a key indistinguishability experiment as:

```
Experiment $\mathbf{Exp}_A^{ind-\kappa}[\Pi,\mathbb{X}]$
    $(\kappa^0,S^0) \overset{R}{\leftarrow} \mathsf{Share}();\quad (\kappa^1,S^1) \overset{R}{\leftarrow} \mathsf{Share}();$
    $b \overset{R}{\leftarrow} \{0,1\};$
    $b' \overset{R}{\leftarrow} A^{\mathsf{corrupt}(S^b,\cdot)}(\kappa^0,\kappa^1,\text{“corrupt”});$
    output ‘1’ if $b=b'$, else ‘0’
```

In this experiment, the adversary receives two secret keys generated by our sharing algorithm, as well as the shares corresponding to one of the keys and must determine to which key they correspond. We define the advantage of adversary $A$ as $\mathbf{Adv}_A^{ind-\kappa}[\Pi,\mathbb{X}] \overset{\triangle}{=} 2\mathsf{Pr}\left[\mathbf{Exp}_A^{ind-\kappa}[\Pi,\mathbb{X}] \Rightarrow 1\right]-1$.

For a generic ECC, if the adversary makes at most $q_u$ corrupt queries, then her total amount of information is upper-bounded by $Q^{q_u}$. Since we model the hash function applied to pre-key $\tilde{\kappa}$ as a random oracle, the adversary's advantage in distinguishing $\kappa^0$ and $\kappa^1$ is bounded above by $\mathbf{Adv}_A^{ind-\kappa}[\Pi,\mathbb{X}] \leq 1/Q^{k-q_u}$. Assuming an encryption algorithm in which key indistinguishability implies ciphertext indistinguishability (e.g., in an ideal cipher model), this bound then translates to the more general sharing of an arbitrary secret. Thus, we have $\mathbf{Adv}_A^{ind}[\Pi,\mathbb{X}] \leq \varepsilon_u \leq 1/Q^{k-q_u}$. This yields Claim 1 from Section 5.3.

### B.2 Proof of Robustness

With a generic linear $(N,K,D)$-ECC, it is possible to recover a message from a codeword with fewer than $D/2$ errors. Thus, as long as the adversary does not corrupt $D/2$ shares, $\varepsilon_r = 0$. Similarly, such a code can recover from $D-1$ erasures; and can also detect up to $D-1$ errors. As discussed in Appendix A, we can deliberately introduce $\zeta$ chaff shares into the ECC to confound the overinformed adversary. This would change are security parameters such that if $q_r < D/2-\zeta$, then $\mathbf{Adv}_A^{rec}[\Pi,\mathbb{X}] = 0 = \varepsilon_r$, and if $q_r \leq D-1-\zeta$, then $\mathbf{Adv}_A^{rec-or-detect}[\Pi,\mathbb{X}] = 0$. This yields Claim 2 from Section 5.3.

## C Proofs of Security and Recoverability for our SWISS Scheme

We prove that our generic family of SWISS schemes from Section 6.2.3 meets our privacy and recoverability requirements. Since our main construction from Section 6.2.2 is a specific instantiation (with $\Psi = 1$), its security follows from the security of the generic family of schemes.

### C.1 Proof of Privacy

To demonstrate that our generic family of SWISS schemes achieves our privacy requirement, we prove Theorem 1 based on the adversary specified in Def. 3. Recall that our generic family of SWISS schemes is parameterized by $\Psi$, one less than the number of overlapping superwindows.

**Proof of Theorem 1:** Suppose we are given an $(\ell, L)$-BDHE instance comprising $\gamma^{(\alpha^i)}$ for $i = 1, 2, ..., L-1$ and the sequence $U' = g'^{(\alpha^i)}$ for $i = 1, 2, ..., \ell - L, \ell + 1, ..., 2\ell$. We construct a SWISS-scheme simulator based on an $(\ell, L, q)$-adversary $A$ as follows.

**Simulator Construction.** First, we construct an appropriate public key by letting $(P_0, P_1, ..., P_{L-1}) = (\gamma, \gamma^{\alpha}, ..., \gamma^{\alpha^{L-1}})$. Then, we select a random $j \in \{1, ..., \ell\}$. This index is our guess as to the superwindow in which the adversary will select a challenge key. If we let $g = g'^{(\alpha^{\ell-j})}$, then $U'$ contains the subsequence $U = g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^{j-L}}, g^{\alpha^{j+1}}, ..., g^{\alpha^{\ell}}$.

We use this subsequence $U$ as the set of underlying superwindow keys in the procedure described in Section 6.2.2, with each superwindow representing a $(k, \frac{\Psi+1}{\Psi} n)$ sharing of $g^{(\alpha^i)}$. For the superwindows corresponding to $g^{(\alpha^{j-L+1})}, ..., g^{(\alpha^j)}$ (which are unknown), we simply share a random value. This procedure creates a set $S$ of shares. If $A$ queries $corrupt(S, i)$, we respond with $S_i$.

To respond to hash queries, we keep a list $\mathcal{V}$ of previous queries. Thus, when $A$ invokes $h(y, z)$ for the first time, we choose a random value $v \overset{R}{\leftarrow} \{0, 1\}^{\tau}$ and add $(y, z, v)$ to the internal list $\mathcal{V}$. If $A$ has previously invoked $h$ on $(y, z)$, then we return the corresponding value of $v$ from $\mathcal{V}$. This creates a perfect implementation of the random oracle contract.

When $A$ terminates, we ignore its output, choose a random hash response $(y, z, v) \overset{R}{\leftarrow} \mathcal{V}$ and return $z$.

**Simulator Correctness.** From the SWISS adversary's point of view, the construction above accurately simulates the *ind-swiss* Experiment. Our replies to the hash queries perfectly instantiate a random oracle, so they offer the adversary no information with which to distinguish a real experiment from a simulation. Our construction deviates from the true protocol in one important respect: the keys for the superwindows corresponding to $g^{(\alpha^{j-L+1})}, ..., g^{(\alpha^j)}$ are chosen at random (since we do not know the appropriate values). However, the definition of $\rho$ precludes the adversary from recovering these superwindow secrets, and hence, she cannot determine that these values do not conform to the expected structure. Nonetheless, because we choose the superwindow secrets at random, we cannot provide the adversary with the correct value of $\kappa_i$. In other words, from our perspective, the value of $\kappa_i$ provided to the adversary is a random value. At some point, the adversary will query $h(r_i, \omega_{kn})$, but since we cannot recognize $\omega_{kn}$, we will not know that we should return $\kappa_i$. Fortunately, by the time the adversary makes this query, we have already extracted the necessary information, namely $\omega_{kn}$, so that even if the adversary quits upon determining a discrepancy, we will still succeed.

**Probability of Success** Our guess $j$ for the superwindow from which $A$ selects a challenge key $\kappa_i$ is correct with probability $\geq 1/\ell$. Since $h$ has a range of $\{0, 1\}^{\tau}$ and $A$ has an $\varepsilon$ advantage, it is clear under the random oracle assumption on $h$ that $A$ inputs $\omega_{jn}$ with probability $\geq \varepsilon - 2^{-\tau}$. If $A$ has queried $h$ with $\omega_{jn}$ in the course of the simulation, then the probability that we output the correct $\omega_{jn} = \hat{e}(g, \gamma)^{(\alpha^{\ell})}$ is just $1/q$.

The only other way the adversary can succeed is by recovering a key for a share she does not hold. However, without the share, the adversary has no knowledge of $r_i$. The random oracle assumption on $h$ guarantees that the adversary succeeds in guessing $\kappa_i$ with probability less than $1/2^{\tau}$. Our theorem bound follows. ■

### C.2 Proof of Recoverability

A legitimate receiver (one who recovers at least $k$ shares out of some window $W'$ of $n$ shares) can determine the key corresponding to each share. Observe that given the overlapping superwindow construction, the window $W'$ is entirely contained within at least one superwindow $W_{\ell n}$. Thus, $k$ elements from $W'$ suffice to reconstruct the superwindow secret $\sigma_{\ell n}$, which can be used to calculate the window secrets $\omega_{\ell n}$, $\omega_{(\ell+1)n}$, ..., $\omega_{(\ell+\Psi)n}$. Each window is of length $n/\Psi$, and hence these two window secrets cover all $(\Psi+1)n/\Psi$ elements in superwindow $W_{\ell n}$. Using the random nonce $r_i$ in each share $S_i$, the legitimate receiver can calculate $\kappa_i$ by hashing $r_i$ with the appropriate window secret.

# CloudAV: N-Version Antivirus in the Network Cloud

*Jon Oberheide, Evan Cooke, Farnam Jahanian*
*Electrical Engineering and Computer Science Department*
*University of Michigan, Ann Arbor, MI 48109*
{jonojono, emcooke, farnam}@umich.edu

## Abstract

Antivirus software is one of the most widely used tools for detecting and stopping malicious and unwanted files. However, the long term effectiveness of traditional host-based antivirus is questionable. Antivirus software fails to detect many modern threats and its increasing complexity has resulted in vulnerabilities that are being exploited by malware. This paper advocates a new model for malware detection on end hosts based on providing antivirus as an in-cloud network service. This model enables identification of malicious and unwanted software by multiple, heterogeneous detection engines in parallel, a technique we term 'N-version protection'. This approach provides several important benefits including *better detection of malicious software*, *enhanced forensics capabilities*, *retrospective detection*, and *improved deployability and management*. To explore this idea we construct and deploy a production quality in-cloud antivirus system called CloudAV. CloudAV includes a lightweight, cross-platform host agent and a network service with ten antivirus engines and two behavioral detection engines. We evaluate the performance, scalability, and efficacy of the system using data from a real-world deployment lasting more than six months and a database of 7220 malware samples covering a one year period. Using this dataset we find that CloudAV provides 35% better detection coverage against recent threats compared to a single antivirus engine and a 98% detection rate across the full dataset. We show that the average length of time to detect new threats by an antivirus engine is 48 days and that retrospective detection can greatly minimize the impact of this delay. Finally, we relate two case studies demonstrating how the forensics capabilities of CloudAV were used by operators during the deployment.

## 1 Introduction

Detecting malicious software is a complex problem. The vast, ever-increasing ecosystem of malicious software and tools presents a daunting challenge for network operators and IT administrators. Antivirus software is one of the most widely used tools for detecting and stopping malicious and unwanted software. However, the elevating sophistication of modern malicious software means that it is increasingly challenging for any single vendor to develop signatures for every new threat. Indeed, a recent Microsoft survey found more than 45,000 new variants of backdoors, trojans, and bots during the second half of 2006 [17].

Two important trends call into question the long term effectiveness of products from a single antivirus vendor. First, there is a significant vulnerability window between when a threat first appears and when antivirus vendors generate a signature. Moreover, a substantial percentage of malware is never detected by antivirus software. This means that end systems with the latest antivirus software and signatures can still be vulnerable for long periods of time. The second important trend is that the increasing complexity of antivirus software and services has indirectly resulted in vulnerabilities that can and are being exploited by malware. That is, malware is actually using vulnerabilities in antivirus software itself as a means to infect systems. SANS has listed vulnerabilities in antivirus software as one of the top 20 threats of 2007 [27].

In this paper we suggest a new model for the detection functionality currently performed by host-based antivirus software. This shift is characterized by two key changes.

1. **Antivirus as a network service:** First, the detection capabilities currently provided by host-based antivirus software can be more efficiently and effectively provided as an *in-cloud network service*. Instead of running complex analysis software on every end host, we suggest that each end host run a lightweight process to detect new files, send them to a network service for analysis, and then permit access or quarantine them based on a report returned

by the network service.

2. **N-version protection:** Second, the identification of malicious and unwanted software should be determined by multiple, heterogeneous detection engines in parallel. Similar to the idea of N-version programming, we propose the notion of N-version protection and suggest that malware detection systems should leverage the detection capabilities of multiple, heterogeneous detection engines to more effectively determine malicious and unwanted files.

This new model provides several important benefits. (1) *Better detection of malicious software*: antivirus engines have complementary detection capabilities and a combination of many different engines can improve the overall identification of malicious and unwanted software. (2) *Enhanced forensics capabilities*: information about what hosts accessed what files provides an incredibly rich database of information for forensics and intrusion analysis. Such information provides temporal relationships between file access events on the same or different hosts. (3) *Retrospective detection*: when a new threat is identified, historical information can be used to identify exactly which hosts or users open similar or identical files. For example, if a new botnet is detected, an in-cloud antivirus service can use the execution history of hosts on a network to identify which hosts have been infected and notify administrators or even automatically quarantine infected hosts. (4) *Improved deployability and management*: Moving detection off the host and into the network significantly simplifies host software enabling deployment on a wider range of platforms and enabling administrators to centrally control signatures and enforce file access policies.

To explore and validate this new antivirus model, we propose an in-cloud antivirus architecture that consists of three major components: a lightweight *host agent* run on end hosts like desktops, laptops, and mobiles devices that identifies new files and sends them into the network for analysis; a *network service* that receives files from hosts and identifies malicious or unwanted content; and an *archival and forensics service* that stores information about analyzed files and provides a management interface for operators.

We construct, deploy, and evaluate a production quality in-cloud antivirus system called CloudAV. CloudAV includes a lightweight, cross-platform host agent for Windows, Linux, and FreeBSD and a network service consisting of ten antivirus engines and two behavioral detection engines. We provide a detailed evaluation of the system using a dataset of 7220 malware samples collected in the wild over a period of a year [20] and a production deployment of our system on a campus network in computer labs spanning multiple departments for a period of over 6 months.

Using the malware dataset, we show how the CloudAV N-version protection approach provides 35% better detection coverage against recent threats compared to a single antivirus engine and 98% detection coverage of the entire dataset compared to 83% with a single engine. In addition, we empirically find that the average length of time to detect new threats by a single engine is 48 days and show how retrospective detection can greatly minimize the impact of this delay.

Finally, we analyze the performance and scalability of the system using deployment results and show that while the total number of executables run by all the systems in a computing lab is quite large (an average of 20,500 per day), the number of unique executables run per day is two orders of magnitude smaller (an average of 217 per day). This means that the caching mechanisms employed in the network service achieves a hit rate of over 99.8%, reducing the load on the network and, in the rare case of a cache miss, we show that the average time required to analyze a file using CloudAV's detection engines is approximately 1.3 seconds.

## 2 Limitations of Antivirus Software

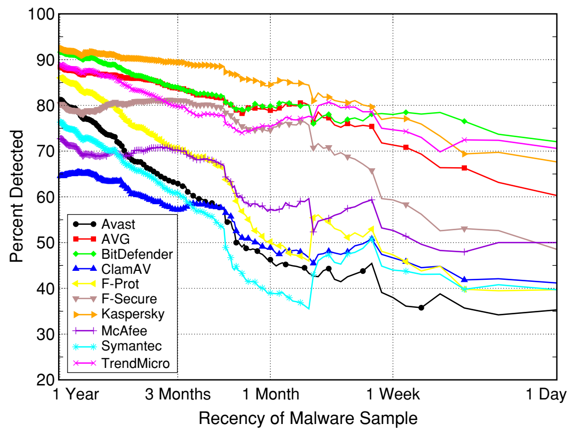
Antivirus software is one of the most successful and widely used tools for detecting and stopping malicious and unwanted software. Antivirus software is deployed on most desktops and workstations in enterprises across the world. The market for antivirus and other security software is estimated to increase to over $10 billion dollars in 2008 [10].

The ubiquitous deployment of antivirus software is closely tied to the ever-expanding ecosystem of malicious software and tools. As the construction of malicious software has shifted from the work of novices to a commercial and financially lucrative enterprise, antivirus vendors must expend more resources to keep up. The rise of botnets and targeted malware attacks for the purposes of spam, fraud, and identity theft present an evolving challenge for antivirus companies. For example, the recent Storm worm demonstrated the use of encrypted peer-to-peer command and control, and the rapid deployment of new variants to continually evade the signatures of antivirus software [4].

However, two important trends call into question the long term effectiveness of products from a single antivirus vendor. The first is that antivirus software fails to detect a significant percentage of malware in the wild. Moreover, there is a significant vulnerability window between when a threat first appears and when antivirus vendors generate a signature or modify their software to detect the threat. This means that end systems with the

| AV Vendor | Version | 3 Months | 1 Month | 1 Week |
|---|---|---|---|---|
| Avast | 4.7.1043 | 62.7% | 45.8% | 39.6% |
| AVG | 7.5.503 | 83.8% | 78.6% | 72.2% |
| BitDefender | 7.1.2559 | 83.9% | 79.7% | 78.5% |
| ClamAV | 0.91.2 | 57.5% | 48.8% | 46.8% |
| CWSandbox | 2.0 | N/A | N/A | N/A |
| F-Prot | 6.0.8.0 | 70.4% | 49.6% | 46.0% |
| F-Secure | 8.00.101 | 80.9% | 74.4% | 60.3% |
| Kaspersky | 7.0.0.125 | 89.2% | 84.0% | 78.5% |
| McAfee | 8.5.0i | 70.5% | 56.7% | 53.9% |
| Norman | 1.8 | N/A | N/A | N/A |
| Symantec | 15.0.0.58 | 60.8% | 38.8% | 45.2% |
| Trend Micro | 16.00 | 79.4% | 74.6% | 75.3% |

(a)



(b)

Figure 1: Detection rate for ten popular antivirus products as a function of the age of the malware samples.

latest antivirus software and signatures can still be vulnerable for long periods of time. The second important trend is that the increasing complexity of antivirus software and services has indirectly resulted in vulnerabilities that can and are being exploited by malware. That is, malware is actually using vulnerabilities in antivirus software as means to infect systems.

## 2.1 Vulnerability Window

The sheer volume of new threats means that it is difficult for any one antivirus vendor to create signatures for all new threats. The ability of any single vendor to create signatures is dependent on many factors such as detection algorithms, collection methodology of malware samples, and response time to 0-day malware. The end result is that there is a significant period of time between when a threat appears and when a signature is created by antivirus vendors (the *vulnerability window*).

To quantify the vulnerability window, we analyzed the detection rate of multiple antivirus engines across malware samples collected over a one year period. The dataset included 7220 samples that were collected between November 11th, 2006 to November 10th, 2007. The malware dataset is described in further detail in Section 6. The signatures used for the antivirus were updated the day after collection ended, November 11th, 2007, and stayed constant through the analysis.

In the first experiment, we analyzed the detection of recent malware. We created three groups of malware: one that included malware collected more recently than 3 months ago, one that included malware collected more recently than 1 month ago, and one that included malware collected more recently than 1 week ago. The antivirus engine and signature versions along with their associated detection rates for each time period are listed in Figure 1(a). The table clearly shows that the detection rate decreases as the malware becomes more recent. Specifically, the number of malware samples detected in the 1 week time period, arguably the most recent and important threats, is quite low.

In the second experiment, we extended this analysis across all the days in the year over which the malware samples were collected. Figure 1(b) shows significant degradation of antivirus engine detection rates as the age, or recency, of the malware sample is varied. As can be seen in the figure, detection rates can drop over 45% when one day's worth of malware is compared to a year's worth. As the plot shows, antivirus engines tend to be effective against malware that is a year old but much less useful in detecting more recent malware, which pose the greatest threat to end hosts.

## 2.2 Antivirus Software Vulnerabilities

A second major concern about the long term viability of host-based antivirus software is that the complexity of antivirus software has resulted in an increased risk of security vulnerabilities. Indeed, severe vulnerabilities have been discovered in the antivirus engines of nearly every vendor. While local exploits are more common (`ioctl` vulnerabilities, overflows in decompression routines, etc), remote exploits in management interfaces have been observed in the wild [30]. Due to the inherent need for elevated privileges by antivirus software, many of these vulnerabilities result in a complete compromise of the affected end host.

Figure 2 shows the number of vulnerabilities reported in the National Vulnerability Database [21] for ten popular antivirus vendors between 2005 and November 2007. This large number of reported vulnerabilities demonstrates not only the risk involved in deploying antivirus

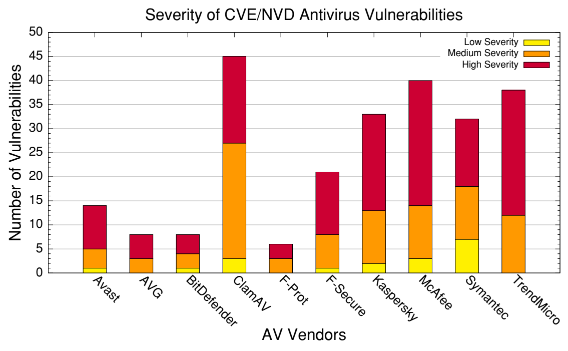


Figure 2: Number of vulnerabilities reported in the National Vulnerability Database (NVD) for ten antivirus vendors between 2005 and November 2007

software, but also an evolution in tactics as attackers are now targeting vulnerabilities in antivirus software itself.

## 3 Approach

This paper advocates a new model for the detection functionality currently performed by antivirus software. First, the detection capabilities currently provided by host-based antivirus software can be more efficiently and effectively provided as an in-cloud network service. Second, the identification of malicious and unwanted software should be determined by multiple, heterogeneous detection engines in parallel.

### 3.1 Deployment Environment

Before getting into details of the approach, it is important to understand the environment in which such an architecture is most effective. First and foremost, we do not see the architecture replacing existing antivirus or intrusion detection solutions. We base our approach on the same threat model as existing host-based antivirus solutions and assume an in-cloud antivirus service would run as an additional layer of protection to augment existing security systems such as those inside an organizational network like an enterprise. Some possible deployment environments include:

- **Enterprise networks:** Enterprise networks tend to be highly controlled environments in which IT administrators control both desktop and server software. In addition, enterprises typically have good network connectivity with low latencies and high bandwidth between workstations and back office systems.
- **Government networks:** Like enterprise networks, government networks tend to be highly controlled with strictly enforced software and security practices. In addition, policy enforcement, access control, and forensic logging can be useful in tracking sensitive information.
- **Mobile/Cellular networks:** The rise of ubiquitous WiFi and mobile 2.5G and 3G data networks also provide an excellent platform for a provider-managed antivirus solution. As mobile devices become increasingly complex, there is an increasing need for mobile security software. Antivirus software has recently become available from multiple vendors for mobile phones [9, 13, 31].

**Privacy implications:** Shifting file analysis to a central location provides significant benefits but also has important privacy implications. It is critical that users of an in-cloud antivirus solution understand that their files may be transferred to another computer for analysis. There are may be situations where this might not be acceptable to users (e.g. many law firms and many consumer broadband customers). However, in controlled environments with explicit network access policies, like many enterprises, such issues are less of a concern. Moreover, the amount of information that is collected can be carefully controlled depending on the environment. As we will discuss later, information about each file analyzed and what files are cached can be controlled depending on the policies of the network.

### 3.2 In-Cloud Detection

The core of the proposed approach is moving the detection of malicious and unwanted files from end hosts and into the network. This idea was originally introduced in [23] and we significantly extend and evaluate the concept in this paper.

There is currently a strong trend toward moving services from end host and monolithic servers into the network cloud. For example, in-cloud email [5, 7, 28] and HTTP [18, 25] filtering systems are already popular and are used to provide an additional layer of security for enterprise networks. In addition, there have been several attempts to provide network services as overlay networks [29, 33].

Moving the detection of malicious and unwanted files into the network significantly lowers the complexity of host-based monitoring software. Clients no longer need to continually update their local signature database, reducing administrative cost. Simplifying the host software also decreases the chance that it could contain exploitable vulnerabilities [15, 30]. Finally, a lightweight host agent allows the service to be extended to mobile and resource-limited devices that lack sufficient processing power but remain an enticing target for malware.

## 3.3 N-Version Protection

The second core component of the proposed approach is a set of heterogeneous detection engines that are used to provide analysis results on a file, also known as *N-version protection*. This approach is very similar to N-version programming, a paradigm in which multiple implementations of critical software are written by independent parties to increase the reliability of software by reducing the probability of concurrent failures [2]. Traditionally, N-version programming has been applied to systems requiring high availability such as distributed filesystems [26]. N-version programming has also been applied to security realm to detect implementation faults in web services that may be exploited by an attacker [19]. While N-version programming uses multiple implementations to increase fault tolerance in complex software, the proposed approach uses multiple independent implementations of detection engines to increase coverage against a highly complex and ever-evolving ecosystem of malicious software.

A few online services have recently been constructed that implement N-version detection techniques. For example, there are online web services for malware submission and analysis [6, 11, 22]. However, these services are designed for the occasional manual upload of a virus sample, rather than the automated and real-time protection of end hosts.

## 4 Architecture

In order to move the detection of malicious and unwanted files from end hosts and into the network, several important challenges must be overcome: (1) unlike existing antivirus software, files must transported into the network for analysis; (2) an efficient analysis system must be constructed to handle the analysis of files from many different hosts using many different detection engines in parallel; and (3) the performance of the system must be similar or better than existing detection systems such as antivirus software.

To address these problems we envision an architecture that includes three major components. The first is a lightweight *host agent* run on end systems like desktops, laptops, and mobiles devices that identifies new files and sends them into the network for analysis. The second is a *network service* that receives files from the host agent, identifies malicious and unwanted content, and instructs hosts whether access to the files is safe. The third component is an *archival and forensics service* that stores information about what files were analyzed and provides a query and alerting interface for operators. Figure 3 shows the high level architecture of the approach.

## 4.1 Client Software

Malicious and unwanted files can enter an organization from many sources. For example, mobile devices, USB drives, email attachments, downloads, and vulnerable network services are all common entry points. Due to the broad range of entry vectors, the proposed architecture uses a lightweight file acquisition agent run on each end system.

Just like existing antivirus software, the host agent runs on each end host and inspects each file on the system. Access to each file is trapped and diverted to a handling routine which begins by generating a unique identifier (UID) of the file and comparing that identifier against a cache of previously analyzed files. If a file UID is not present in the cache then the file is sent to the in-cloud network service for analysis.

To make the analysis process more efficient, the architecture provides a method for sending a file for analysis as soon as it is written on the end host's filesystem (e.g., via file-copy, installation, or download). Doing so amortizes the transmission and analysis cost over the time elapsed between file creation and system or user-initiated access.

### 4.1.1 Threat Model

The threat model for the host agent is similar to that of existing software protection mechanisms such as antivirus, host-based firewalls, and host-based intrusion detection. As with these host-based systems, if an attacker has already achieved code execution privileges, it may be possible to evade or disable the host agent. As described in Section 2, antivirus software contains many vulnerabilities that can be directly targeted by malware due to its complexity. By reducing the complexity of the host agent by moving detection into the network, it is possible to reduce the vulnerability footprint of host software that may lead to elevated privileges or code execution.

### 4.1.2 File Unique Identifiers

One of the core components of the host agent is the file unique identifier (UID) generator. The goal of the UID generator is to provide a compact summary of a file. That summary is transmitted over the network to determine if an identical file has already been analyzed by the network service. One of the simplest methods of generating such a UID is a cryptographic hash of a file, such as MD5 or SHA-1. Cryptographic hashes are fast and provide excellent resistance to collision attacks. However, the same collision resistance also means that changing a single byte in a file results in completely different UID. To combat polymorphic threats, a more complex UID generator algorithm could be employed. For example,

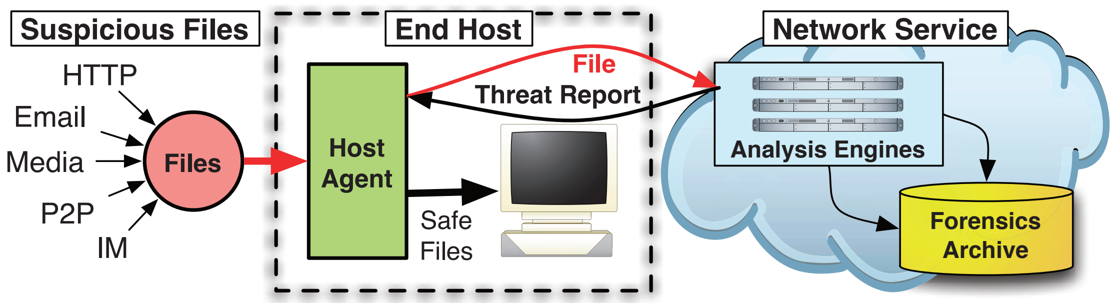


Figure 3: Architectural approach for in-cloud file analysis service.

a method such as locality-preserving hashing in multi-dimensional spaces [12] could be used track differences between two files in a compact manner.

### 4.1.3 User Interface

We envision three majors modes of operation that affect how users interact with the host agent that range from less to more interactive.

- **Transparent mode:** In this mode, the detection software is completely transparent to the end user. Files are sent into the cloud for analysis but the execution or loading of a file is never blocked or interrupted. In this mode end hosts can become infected by known malware but administrators can use detection alerts and detailed forensic information to aid in cleaning up infected systems.

- **Warning mode:** In this mode, access to a file is blocked until an *access directive* has been returned to the host agent. If the file is classified as unsafe then a warning is presented to the user instructing them why the file is suspicious. The user is then allowed to make the decision of whether to proceed in accessing the file or not.

- **Blocking mode:** In this mode, access to a file is blocked until an *access directive* has been returned to the host agent. If the file is classified as suspicious then access to the file is denied and the user is informed with an error dialog.

### 4.1.4 Other File Acquisition Methods

While the host agent is the primary method of acquiring candidate files and transmitting them to the network service for analysis, other methods can also be employed to increase the performance and visibility of the system. For example, a network sensor or tap monitoring the traffic of a network may pull files directly out of a network stream using deep packet inspection (DPI) techniques. By identifying files and performing analysis before the file even reaches the destination host, the need to retransmit the file to the network service is alleviated and user-perceived latencies can be reduced. Clearly this approach cannot completely replace the host agent as network traffic can be encrypted, files may be encapsulated in unknown protocols, and the network is only one source of malicious content.

## 4.2 Network Service

The second major component of the architecture is the network service responsible for file analysis. The core task of the network service is to determine whether a file is malicious or unwanted. Unlike existing systems, each file is analyzed by a collection of detection engines. That is, each file is analyzed by multiple detection engines in parallel and a final determination of whether a file is malicious or unwanted is made by aggregating these individual results into a threat report.

### 4.2.1 Detection Engines

A cluster of servers can quickly analyze files using multiple detection techniques. Additional detection engines can easily be integrated into a network service, allowing for considerable extensibility. Such comprehensive analysis can significantly increase the detection coverage of malicious software. In addition, the use of engines from different vendors using different detection techniques means that the overall result does not rely too heavily on a single vendor or detection technology.

A wide range of both lightweight and heavyweight detection techniques can be used in the backend. For example, lightweight detection systems like existing *antivirus engines* can be used to evaluate candidate files. In addition, more heavyweight detectors like *behavioral analyzers* can also be used. A behavioral system executes a suspicious file in a sandboxed environment (e.g., Norman

Sandbox [22], CWSandbox [6]) or virtual machine and records host state changes and network activity. Such deep analysis is difficult or impossible to accomplish on resource-constrained devices like mobile phones but is possible when detection is moved to dedicated servers. In addition, instead of forcing signature updates to every host, detection engines can be kept up-to-date with the latest vendor signatures at a central source.

Finally, running multiple detection engines within the same service provides the ability to correlation information between engines. For example, if a detector finds that the behavior of an unknown file is similar to that of an file previously classified as malicious by antivirus engines, the unknown file can be marked as suspicious [23].

#### 4.2.2 Result Aggregation

The results from the different detection engines must be combined to determine whether a file is safe to open, access, or execute. Several variables may impact this process.

First, results from the detection engines may reach the aggregator at different times – if a detector fails, it may never return any results. In order to prevent a slow or failed detector from holding up a host, the aggregator can use a subset of results to determine if a file is safe. Determining the size of such a quorum depends on the deployment scenario and variables like the number of detection engines, security policies, and latency requirements.

Second, the metadata returned by each detector may be different so the detection results are wrapped in a container object that describes how the data should be interpreted. For example, behavioral analysis reports may not indicate whether a file is safe but can be attached to the final aggregation report to help users, operators, or external programs interpret the results.

Lastly, the threshold at which a candidate file is deemed unsafe or malicious may be defined by security policy of the network's administrators. For example, some administrators may opt for a strict policy where a single engine is sufficient to deem a file malicious while less security-conscious administrators may require multiple engines to agree to deem a file malicious. We discuss the balance between coverage and confidence further in Section 7.

The result of the aggregation process is a threat report that is sent to the host agent and can be cached on the server. A threat report can contain a variety of metadata and analysis results about a file. The specific contents of the report depend on the deployment scenario. Some possible report sections include: (1) an operation directive; a set of instructions indicating the action to be performed by the host agent, such as how the file should be accessed, opened, executed, or quarantined; (2) family/variant labels; a list of malware family/variant classification labels assigned to the file by the different detection engines; and (3) behavioral analysis; a list of host and network behaviors observed during simulation. This may include information about processes spawned, files and registry keys modified, network activity, or other state changes.

#### 4.2.3 Caching

Once a threat report has been generated for a candidate file, it can be stored in both a local cache on the host agent and in a shared remote cache on the server. This means that once a file has been analyzed, subsequent accesses to that file by the user can be determined locally without requiring network access. Moreover, once a single host in a network has accessed a file and sent it to the network service for analysis, any subsequent access of the same file by other hosts in the network can leverage the existing threat report in the shared remote cache on the server. Cached reports stored in the network service may also periodically be *pushed* to the host agent to speed up future accesses and invalidated when deemed necessary.

## 4.3 Archival and Forensics Service

The third and final component of the architecture is a service that provides information on file usage across participating hosts which can assist in post-infection forensic analysis. While some forensics tracking systems [14, 8] provide fine-grained details tracing back to the exact vulnerable processes and system objects involved in an infection, they are often accompanied by high storage requirements and performance degradation. Instead, we opt for a lightweight solution consisting of file access information sent by the host agent and stored securely by the network service, in addition to the behavioral profiles of malicious software generated by the behavioral detection engines. Depending on the privacy policy of organization, a tunable amount of forensics information can be logged and sent to the archival service. For example, a more security conscious organization could specify that information about every executable launch be recorded and sent to the archival service. Another policy might specify that only accesses to unsafe files be archived without any personally identifiable information.

Archiving forensic and file usage information provides a rich information source for both security professionals and administrators. From a security perspective, tracking the system events leading up to an infection can assist in determining its cause, assessing the risk involved with the compromise, and aiding in any necessary disinfection and cleanup. In addition, threat reports from behavioral engines provide a valuable source of forensic data as the exact operations performed by a piece of malicious software can be analyzed in detail. From a general administration perspective, knowledge of what applications and files are frequently in use can aid the placement of file caches, application servers, and even be used to determine the optimal number of licenses needed for expensive applications.

Consider the outbreak of a zero-day exploit. An enterprise might receive a notice of a new malware attack and wonder how many of their systems were infected. In the past, this might require performing an inventory of all systems, determining which were running vulnerable software, and then manually inspecting each system. Using the forensics archival interface in the proposed architecture, an operator could search for the UID of the malicious file over the past few months and instantly find out where, when, and who opened the file and what malicious actions the file performed. The impacted machines could then immediately be quarantined.

The forensics archive also enables *retrospective detection*. The complete archive of files that are transmitted to the network service may be re-scanned by available engines whenever a signature update occurs. Retrospective detection allows previously undetected malware that has infected a host to be identified and quarantined.

## 5 CloudAV Implementation

To explore and validate the proposed in-cloud antivirus architecture, we constructed a production quality implementation called CloudAV. In this section we describe how CloudAV implements each of the three main components of the architecture.

### 5.1 Host Agent

We implement the host agent for a variety of platforms including Windows 2000/XP/Vista, Linux 2.4/2.6, and FreeBSD 6.0+. The implementation of the host agent is designed to acquire executable files for analysis by the in-cloud network service, as executables are a common source of malicious content. We discuss how the agent can be extended to acquire DLLs, documents, and other common malcode-bearing files types in Section 7.

While the exact APIs are platform dependent (CreateProcess on Win32, execve syscall on Linux 2.4, LSM hooks on Linux 2.6, etc), the host agent hooks and interposes on system events. This interposition is implemented via the MadCodeHook [16] package on the Win32 platform and via the Dazuko [24] framework for the other platforms. Process creation events are interposed upon by the host agent to acquire and process candidate executables before they are allowed to continue. In addition, filesystem events are captured to identify new files entering a host and preemptively transfer them to the network service before execution to eliminate any user-perceived latencies.

As motivating factors of our work include the complexity and security risks involved in running host-based antivirus, the host agent was designed to be simple and lightweight, both in code size and resource requirements. The Win32 agent is approximately 1500 lines of code of which 60% is managed code, further reducing the vulnerability profile of the agent. The agent for the other platforms is written in python and is under 300 lines of code.

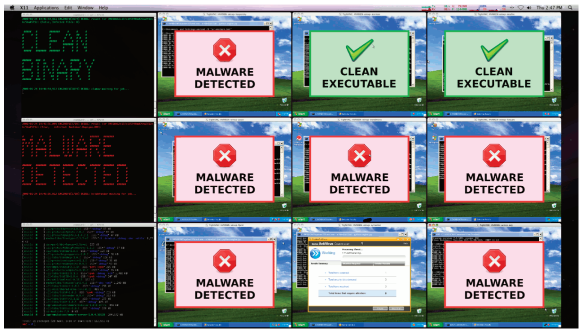
While the host agent is primarily targeted at end hosts, our architecture is also effective in other deployment scenarios such as mail servers. To demonstrate this, we also implemented a milter (mail filter) frontend for use with mail transfer agents (MTAs) such as Sendmail and Postfix to scan all attachments on incoming emails. Using the pymilter API, the milter frontend weighs in at approximately 100 lines of code.

### 5.2 Network Service

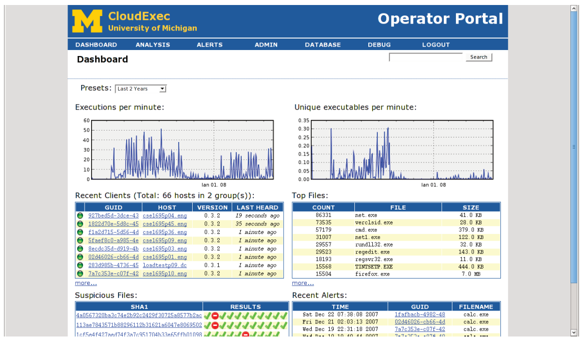
The network service acts as a dispatch manager between the host agent and the backend analysis engines. Incoming candidate files are received, analyzed, and a threat report is returned to the host agent dictating the appropriate action to take. Communication between the host agent and the network service uses a HTTP wire protocol protected by mutually authenticated SSL/TLS. Between the components within the network service itself, communication is performed via a publish/subscribe bus to allow modularization and effective scalability.

The network service allows for various priorities to be assigned to analysis requests to aid latency-sensitive applications and penalize misbehaving hosts. For example, application and mail scanning may take higher analysis priority than background analysis tasks such as retroactive detection (described in Section 7). This also enables the system to penalize or temporarily suspend misbehaving hosts than may try to submit many analysis requests or otherwise flood the system.

Each backend engine runs in a Xen virtualized container, which offers significant advantages in terms of isolation and scalability. Given the numerous vulnerabilities in existing antivirus software discussed in Section 2, isolation of the antivirus engines from the rest of the system is vital. If one of the antivirus engines in the backend is targeted and successfully exploited by a malicious candidate file, the virtualized container can simply be disposed of and immediately reverted to a clean snapshot. As for scalability, virtualized containers allows the network service to spin up multiple instances of a partic-



(a)



Dashboard

Presets: Last 2 Years

Executions per minute:

Unique executables per minute:

Recent Clients (Total: 66 hosts in 2 group(s)):

| | GUID | HOST | VERSION | LAST HEARD |
|---|---|---|---|---|
| ● | 927bed5d-3dce-43 | cse1695p04.eng | 0.3.2 | 19 seconds ago |
| ● | 1822d70e-5d8c-45 | cse1695p45.eng | 0.3.2 | 35 seconds ago |
| ● | f1a2d715-5d56-4d | cse1695p36.eng | 0.3.2 | 1 minute ago |
| ● | 5faef8c0-a985-4e | cse1695p09.eng | 0.3.2 | 1 minute ago |
| ● | 8ecdc35d-d919-4b | cse1695p03.eng | 0.3.2 | 1 minute ago |
| ● | 02d46026-cb66-4d | cse1695p01.eng | 0.3.2 | 1 minute ago |
| ● | 283d985b-4736-45 | loadtestp09.dc | 0.3.1 | 1 minute ago |
| ● | 7a7c353e-c07f-42 | cse1695p10.eng | 0.3.2 | 1 minute ago |

Top Files:

| COUNT | FILE | SIZE |
|---|---|---|
| 86331 | net.exe | 41.0 KB |
| 73535 | verclsid.exe | 28.0 KB |
| 57179 | cmd.exe | 379.0 KB |
| 31007 | net1.exe | 122.0 KB |
| 29557 | rundll32.exe | 32.0 KB |
| 29523 | regedit.exe | 143.0 KB |
| 18193 | regsvr32.exe | 11.0 KB |
| 15568 | TINTSETP.EXE | 444.0 KB |
| 15504 | firefox.exe | 7.0 MB |

Suspicious Files:

| SHA1 | RESULTS |
|---|---|
| 4a0567320ba3c74e2b92c2429f30725a8577b2ac | ✓⛔✓✓✓✓✓✓✓✓✓ |
| 113ae7843571b88296112b31621a6047e8069502 | ✓⛔✓✓✓✓✓✓✓✓✓ |
| 1cf5e4f427aed74f3a7c951704b33e65ffb01098 | ✓✓✓✓✓⛔✓✓✓✓ |

Recent Alerts:

| TIME | GUID | FILENAME |
|---|---|---|
| Sat Dec 22 07:38:08 2007 | 1fafbacb-4982-48 | calc.exe |
| Fri Dec 21 02:03:13 2007 | 02d46026-cb66-4d | calc.exe |
| Wed Dec 19 22:31:18 2007 | 7a7c353e-c07f-42 | calc.exe |

(b)

Figure 4: Screen captures of the detection engine VM monitoring interface (a) and the web management portal which provides access to forensic data and threat reports (b).

ular engine when demand for its services increase.

Our current implementation employs 12 engines: 10 traditional antivirus engines (Avast, AVG, BitDefender, ClamAV, F-Prot, F-Secure, Kaspersky, McAfee, Symantec, and Trend Micro) and 2 behavioral engines (Norman Sandbox and CWSandbox). The exact version of each detection engine is listed in Figure 1(a). 9 of the backend engines run in a Windows XP environment using Xen's HVM capabilities while the other 3 run in a Gentoo Linux environment using Xen domU paravirtualization. Implementing each particular engine for the backend is a simple task and extending the backend with additional engines in the future is equally as simple. For reference, the amount of code required for each engine is 42 lines of python code on average with a median of 26 lines of code.

### 5.3 Management Interface

The third component is a management interface which provides access to the forensics archive, policy enforcement, alerting, and report generation. These interfaces are exposed to network administrators via a web-based management interface. The web interface is implemented using Cherrypy, a python web development framework. A screen capture of the dashboard of the management interface is depicted in Figure 4.

The centralized management and network-based architecture allows for administrators to enforce network-wide policies and define alerts when those policies are violated. Alerts are defined through a flexible specification language consisting of attributes describing an access request from the host agent and boolean predicates similar to an SQL WHERE clause. The specification language allows for notification for triggered alerts (via email, syslog, SNMP) and enforcement of administrator-defined policies.

For example, network administrators may desire to block certain applications from being used on end hosts. While these unwanted applications may not be explicitly malicious, they may have a negative effect on host or network performance or be against acceptable use policies. We observed several classes of these potentially unwanted applications in our production deployment including P2P applications (uTorrent, Limewire, etc) and multi-player gaming (World of Warcraft, online poker, etc). Other policies can be defined to reinforce prudent security practices, such as blocking the user from executing attachments from an email application.

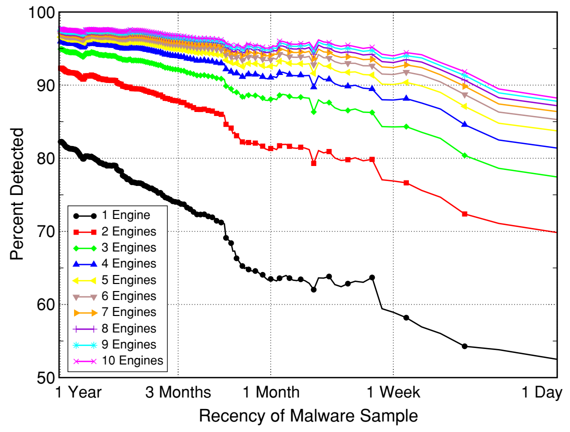
## 6 Evaluation

In this section, we provide an evaluation of the proposed architecture through two distinct sources of data. The first source is a dataset of malicious software collected over a period of a year. Using this dataset, we evaluate the effectiveness of N-version protection and retrospective detection. We also utilize this malware dataset to empirically quantify the size of vulnerability window.

The second data source is derived from a production deployment of the system on a campus network in computer labs spanning multiple departments for a period of over 6 months. We use the data collected from this deployment to explore the performance characteristics of CloudAV. For example, we analyze the number of files handled by the network service, the utility of the caching system, and the time it takes the detection engines to analyze individual files. In addition, we use deployment data to demonstrate the forensics capabilities of the approach. We detail two real-world case studies from the deployment, one involving an infection by malicious software and one involving a suspicious, yet legitimate executable.

| Engines | 3 Months | 1 Month | 1 Week |
|---|---|---|---|
| 1 | 73.9% | 63.1% | 59.6% |
| 2 | 87.7% | 81.0% | 77.6% |
| 3 | 92.0% | 87.8% | 84.8% |
| 4 | 93.8% | 90.9% | 88.4% |
| 5 | 94.8% | 92.4% | 90.5% |
| 6 | 95.4% | 93.4% | 91.8% |
| 7 | 95.9% | 94.0% | 92.8% |
| 8 | 96.2% | 94.5% | 93.5% |
| 9 | 96.5% | 94.8% | 94.0% |
| 10 | 96.7% | 95.0% | 94.4% |

(a)



(b)

Figure 5: The average detection coverage for the various datasets (a) and the continuous coverage over time (b) when a given number of engines are used in parallel.

## 6.1 Malware Dataset Results

The first component of the evaluation is based on a malware dataset obtained through Arbor Network's Arbor Malware Library (AML) [20]. AML is composed of malware collected using a variety of techniques such as distributed darknet honeypots, spam traps, and honeyclient spidering. The use of a diverse set of collection techniques means that the malware samples are more representative of threats faced by end hosts than malware datasets collected using only a single collection methodology such as Nepenthes [3]. The AML dataset used in this paper consists of 7220 unique malware samples collected over a period of one year (November 12th, 2006 to November 11th, 2007). An average of 20 samples were collected each day with a standard deviation of 19.6 samples.

### 6.1.1 N-Version Protection

We used the AML malware dataset to assess the effectiveness of a set of heterogeneous detection engines. Figure 5(a) and (b) show the overall detection rate across different time ranges of malware samples as the number of detection engines is increased. The detection rates were determined by looking at the average performance across all combinations of N engines for a given N. For example, the average detection rate across all combinations of two detection engines over the most recent 3 months of malware was 87.7%.

Figure 5(a) demonstrates how the use of multiple heterogeneous engines allows CloudAV to significantly improve the aggregate detection rate. Figure 5(b) shows the detection rate over malware samples ranging from one day old to one year old. The graph shows how using ten engines can increase the detection rate for the entire year-long AML dataset as high as 98%.

The graph also reveals that CloudAV significantly improves the detection rate of more recent malware. When a single antivirus engine is used, the detection rate degrades from 82% against a year old dataset to 52% against a day old dataset (a decrease of 30%). However, using ten antivirus engines the detection coverage only goes from 98% down to 88% (a decrease of only 10%). These results show that not only do multiple engines complement each other to provide a higher detection rate, but the combination has resistance to coverage degradation as the encountered threats become more recent. As the most recent threats are typically the most important, a detection rate of 88% versus 52% is a significant advantage.

Another noticeable feature of Figure 5 is the decrease in incremental coverage. Moving from 1 to 2 engines results in a large jump in detection rate, moving from 2 to 3 is smaller, moving from 3 to 4 is even smaller, and so on. The diminishing marginal utility of additional engines shows that a practical balance may be reached between detection coverage and licensing costs, which we discuss further in Section 7.

In addition to the averages presented in Figure 5, the minimum and maximum detection coverage for a given number of engines is of interest. For the one week time range, the maximum detection coverage when using only a single engine is 78.6% (Kaspersky) and the minimum is 39.7% (Avast). When using 3 engines in parallel, the maximum detection coverage is 93.6% (BitDefender, Kaspersky, and Trend Micro) and the minimum is 69.1% (ClamAV, F-Prot, and McAfee). However, the optimal combination of antivirus vendors to achieve the most comprehensive protection against malware may not be a simple measure of total detection coverage. Rather, a number of complex factors may influence the best choice of detection engines, including the types of threats most commonly faced by the hosts being protected, the algorithms used for detection by a particular vendor, the vendor's response time to 0-day malware, and the collection methodology and visibility employed by the vendor to collect new malware.

## 6.2 Retrospective Detection

We also used the AML malware dataset to understand the utility of retrospective detection. Recall that retrospective detection is the ability to use historical information and archived files stored by CloudAV to retrospectively detect and identify hosts infected that with malware that has previously gone undetected. Retrospective detection is an especially important post-infection defense against 0-day threats and is independent of the number or vendor of antivirus engines employed. Imagine a polymorphic threat not detected by any antivirus or behavioral engine that infects a few hosts on a network. In the host-based antivirus paradigm, those hosts could become infected, have their antivirus software disabled, and continue to be infected indefinitely.

In the proposed system, the infected file would be sent to the network service for analysis, deemed clean, archived at the network service, and the host would become infected. Then, when any of the antivirus vendors update their signature databases to detect the threat, the previously undetected malware can be re-scanned in the network service's archive and flagged as malicious. Instantly, armed with this new information, the network service can identify which hosts on the network have been infected in the past by this malware from its database of execution history and notify the administrators with detailed forensic information.

Retrospective detection is especially important as frequent signature updates from vendors continually add coverage for previously undetected malware. Using our AML dataset and an archive of a year's worth of McAfee DAT signature files (with a one week granularity), we determined that approximately 100 new malware samples were detected each week on average (with a standard deviation of 57) by the McAfee updates. More importantly, for those samples that were eventually detected by a signature update (5147 out of 7220), the average time from when a piece of malware was observed to when it was detected (i.e. the vulnerability window) was approximately 48 days. A cumulative distribution function of the days between observation and detection is depicted in Figure 6.

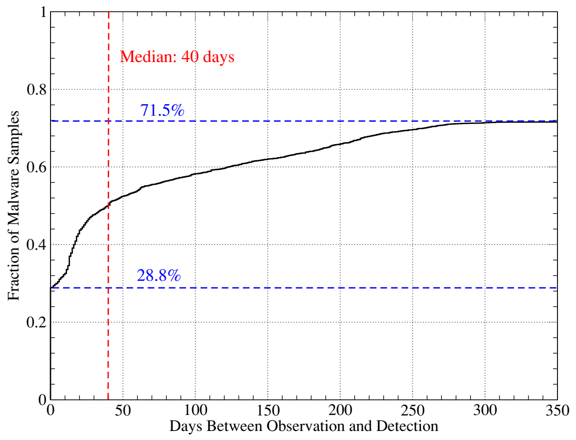


Figure 6: Cumulative distribution function depicting the number of days between when a malware sample is observed and when it is first detected by the McAfee antivirus engine.

## 6.3 Deployment Results

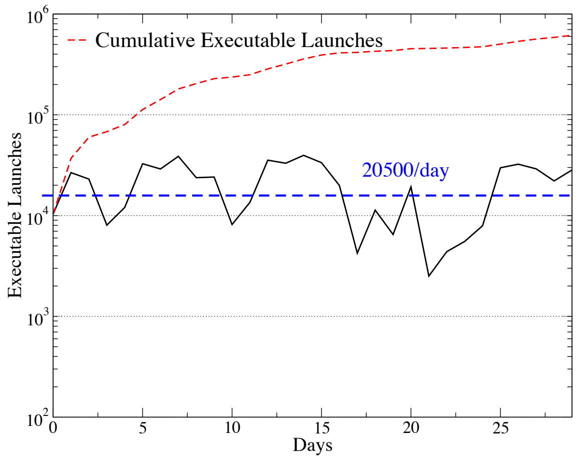
With the aid of network operations and security staff we deployed CloudAV across a large campus network. In this section, we discuss results based on the data collected as a part of this deployment.

### 6.3.1 Executable Events

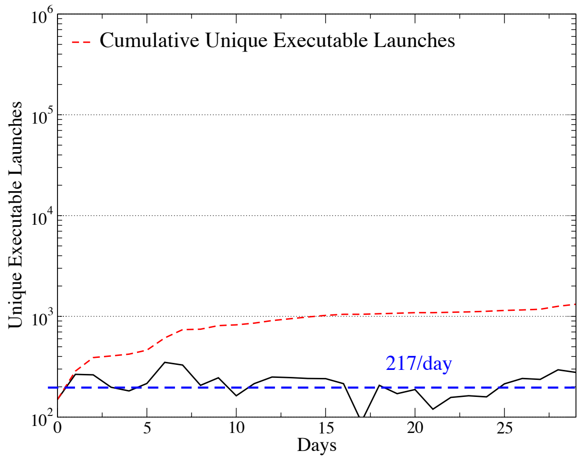
One of the core variables that impacts the resource requirements of the network service is the rate at which new files must be analyzed. If this rate is extremely high, extensive computing resources will be required to handle the analysis load. Figure 7 shows the number of total execution events and unique executables observed during a one month period in a university computing lab.

Figure 7 shows that while the total number of executables run by all the systems in the lab is quite large (an average of 20,500 per day), the number of unique executables run per day is two orders of magnitude smaller (an average of 217 per day). Moreover, the number of unique executables is likely inflated due to the fact that these machines are frequently used by students to work on computer science class projects, resulting in a large number of distinct executables with each compile of a project. A more static, non-development environment would likely see even less unique executables.

We also investigated the origins of these executables based on the file path of 1000 unique executables stored in the forensics archive. Table 1 shows the break down of these sources. The majority of executables originate from the local hard drive but a significant portion were launched from various network sources. Executables from the temp directory often indicate that they were



(a)



(b)

Figure 7: Executable launches (a) and unique executable launches (b) per day over a one month period in a representative sample of 50 machines in the deployment.

| | | |
|---|---|---|
| **Local Drives** 52.4% | Program Files | 22.3% |
| | Temp Directory | 14.2% |
| | Windows Directory | 13.4% |
| | Other | 2.4% |
| **Network Drives** 43.3% | Engineering Apps | 23.6% |
| | User Desktop Shares | 9.3% |
| | User AFS Shares | 8.3% |
| | Other | 2.1% |
| **External Media** 4.4% | USB Flash | 2.4% |
| | CDROM Drive | 2.0% |

Table 1: A distribution of the sources of 1000 executables observed in during the deployment of our host agent over a six-month period.

downloaded via a web browser and executed, contributing even more to networked origins. In addition, a nontrivial number of executables were introduced to the system directly from external media such as a CDROM drive and USB flash media. This diversity exemplifies the need for a host agent that is capable of acquiring files from a variety of sources.

### 6.3.2 Caching and Performance

A second important variable that determines the scalability and performance of the system is the cache hit rate. A hit in the local cache can prevent network requests, and a hit in the remote cache can prevent unnecessary files transfers to the network service. The hosts instrumented as a part of the deployment were heavily loaded Windows XP workstations. The Windows Start Menu contained over 250 executable applications including a wide range of internet, multimedia, and engineering packages.

Our results indicate that 10 processes were launched from when the host agent service loads to when the login screen appears and another 52 processes were launched before the user's desktop loaded. As a measure of overhead, we measured the number of bytes transferred between a specific client and network service under different caching conditions. With a warm remote cache, the boot-up process took 8.7 KB and the login process took 46.2 KB. In the case of a cold remote cache, which would only ever occur a single time when the first host in the network loaded for the first time, the boot-up process took 406 KB and the login process took 12.5 MB. For comparison, the Active Directory service installed on the deployment machines took 171 KB and 270 KB on boot and login respectively.

It is also possible to evaluate the performance of the caching system by looking at Figure 7. We recorded almost over 615,000 total execution events over one month yet only observed 1300 unique executables. As a remote cache miss only happens when a new executable is observed, the remote cache hit rate is approximately 99.8%. Even more significant, the local cache can be pre-seeded with known installed software during the host agent installation process, improving the hit rate further. In the infrequent case when a miss occurs in both the local and remote cache, the candidate file must be transferred to the network service. Network latency, throughput, and analysis time all affect the user-perceived delay between when a file is acquired by the host agent and a threat report is returned by the network service. As local networks usually have low latencies and high bandwidth, the analysis time of files will often dominate the network latency and throughput delay. The average time for a detection engine to analyze a candidate file in the AML dataset was approximately 1.3 seconds with a standard deviation of 1.8 seconds.

#### 6.3.3 Forensics Case Studies

We review two case studies from the deployment concerning two real-world events that demonstrate the utility of the forensics archive.

**Malware Case Study:** While running the host agent in transparent mode in the campus deployment, the CloudAV system alerted us to a candidate executable that had been marked as malicious by multiple antivirus engines. It is important to note that this malicious file successfully evaded the local antivirus software (McAfee) that was installed along side our host agent. Immediately, we accessed the management interface to view the forensics information associated with the tracked execution event and runtime behavioral results provided the two behavioral engines employed in our network service.

The initial executable launched by the user was `warcraft3keygen.exe`, an apparent serial number generator for the game Warcraft 3. This executable was just a bootstrap for the `m222.exe` executable which was written to the Windows temp directory and subsequently launched via CreateProcess. `m222.exe` then copied itself to `C:\Program Files\Intel\Intel`, made itself hidden and read-only, and created a fraudulent Windows service via the Service Control Manager (SCM) called `Remote Procedure Call (RPC) MO` to launch itself automatically at system startup. Additionally, the malware attempted to contact command and control infrastructure through DNS requests for several names including `50216.ipread.com`, but the domains had already been blackholed.

**Legitimate Case Study:** In another instance, we were alerted to a candidate executable that was flagged as suspicious by several engines. The executable in question was the PsExec utility from SysInternals which allows for remote control and command execution. Given that this utility can be used for both malicious and legitimate purposes, it was worthy of further investigation to determine its origin.

Using the management interface, we were able to immediately drill down to the affected host, user, files, and environment of the suspected event. The PsExec service `psexesvc.exe` was first launched from the parent process `services.exe` when an incoming remote execution request arrived from the PsExec client. The next execution event was `net.exe` with the command line argument `localgroup administrators`, which results in the listing of all the users in the local administrators group.

Three factors led us to dismiss the event as legitimate. First, the operation performed by the net command was not overtly malicious. Second, the user performing this action was a known network administrator. Lastly, we were able to determine the net.exe executable was identical to the one deployed across all the hosts in the network, ruling out the case where the net.exe program itself may have been a trojaned version. While this event could be seen as a false positive, it is actually an important alert that needs to dealt with by a network administrator. The forensic and historical information provided through the management interface allows these events to be dealt with remotely in an accurate and efficient manner.

## 7 Discussion and Limitations

Moving detection functionality into the network cloud has other technical and practical implications. In this section we attempt to highlight limitations of the proposed model and then describe a few resulting benefits.

### 7.1 User Context and Environment in Detection Engines

One important benefit of running detection engines on end systems is that local context such as user input, network input, operating system state, and the local filesystem are available to aid detection algorithms. For example, many antivirus vendors use behavioral detection routines that monitor running processes to identify misbehaving or potentially malicious programs.

While it is difficult to replicate the entire state of end systems inside the network cloud, there are two general techniques an in-cloud antivirus system can use to provide additional context to detection engines. First, detection engines can open or execute files inside a VM instance. For example, existing antivirus behavioral detection system can be leveraged by opening and running files inside a virtual antivirus detection instance. A second technique is to replicate more of the local end system state in the cloud. For example, when a file is sent to the network service, contextual metadata such as other running processes can be attached to the submission and used to aid detection. However, because complete local state can be quite large, there are many instances where deploying local detection agents may be required to compliment in-cloud detection.

## 7.2 Disconnected Operation

Another challenge with moving detection into the network is that network connectivity is needed to analyze files. An end host participating in the service may enter a disconnected state for many reasons including network outages, mobility constraints, misconfiguration, or denial of service attacks. In such a disconnected state, the host agent may not be able to reach the network service to check the remote cache or to submit new files for analysis. Therefore, in certain scenarios, the end host may be unable to complete its desired operations.

Addressing the issue of disconnected operation is primarily an issue of policy, although the architecture includes technical components that aid in continued protection in a disconnected state. For example, the local caching employed by our host agent effectively allows a disconnected user to access files that have previously been analyzed by the network service. However, for files that have not yet been analyzed, a policy decision is necessary. Security-conscious organizations may select a strict policy requiring that users have network connectivity before accessing new applications, while organizations with less strict security policies may desire more flexibility. As our host agent works together with host-based antivirus, local antivirus software installed on the end host may provide adequate protection for these environments with more liberal security policies until network access is restored.

## 7.3 Sources of Malicious Behavior

Malicious code or inputs that cause unwanted program behavior can be present in many places such as in the linking, loading, or running of the initial program instructions, and the reading of input from memory, the filesystem, or the network. For example, some types of malware use external files such as DLLs loaded at runtime to store and later execute malicious code. In addition, recent vulnerabilities in desktop software such as Adobe Acrobat [1] and Microsoft Word [32] have exemplified the threat from documents, multimedia, and other non-executable malcode-bearing file types. Developing a host agent that handles all these different sources of malicious behavior is challenging.

The CloudAV implementation described in this paper focuses on executables, but the host agent can be extended to identify other file types. To explore the challenges of extending the system we modified the host agent to monitor the DLL dependencies for each executable acquired by the host agent. Each dependent DLL of an application is processed similar to the executable itself: the local and remote cache is checked to determine if it has been previously analyzed, and if not, it is transmitted to the network service for analysis. Extending the host agent further to handle documents would be as simple as instructing the host agent to listen for filesystem events for the desired file types. In fact, the types of files acquired by the host agent could be dynamically configured at a central location by an administrator to adapt to evolving threats.

| AV Vendor | 3 Months | 1 Month | 1 Week |
|---|---|---|---|
| Avast | +14.8% | +16.6% | +24.6% |
| AVG | +5.9% | +6.8% | +8.7% |
| BitDefender | +4.0% | +5.3% | +3.1% |
| ClamAV | +0.0% | +0.0% | +0.0% |
| F-Prot | +9.9% | +15.3% | +12.6% |
| F-Secure | +7.9% | +9.3% | +15.0% |
| Kaspersky | +1.5% | +1.9% | +2.3% |
| McAfee | +10.6% | +14.0% | +14.2% |
| Symantec | +17.8% | +23.0% | +20.6% |
| Trend Micro | +9.8% | +11.5% | +12.6% |

Table 2: The percentage increase in detection coverage obtained when ClamAV, a truly free engine, is added to a deployment with only a single engine.

## 7.4 Detection Engine Licensing

Most of the antivirus and behavioral engines employed in our architecture required paid licenses. Acquiring licenses for all the engines may be infeasible for some organizations. While we have chosen a large number of engines for evaluation and measurement purposes, the full amount may not be necessary to obtain effective protection. As seen in Figure 5, ten engines may not be the most effective price/performance point as diminishing returns are observed as more engines are added.

We currently employ four free engines in our system for which paid licenses were not necessary: AVG, Avast, BitDefender, and ClamAV. Using only these four engines, we are still able to obtain 94.3%, 92.0%, and 88.0% detection coverage over periods of 3 months, 1 month, and 1 week respectively. These detection coverage values for the combined free engines exceed every single vendor in each dataset period.

While the interpretation of the various antivirus licenses is unclear in our architecture, especially with regards to virtualization, it is likely that site-wide licenses would be needed for the “free” engines for a commercial deployment. Even if only one licensed engine is used, our system still maintains the benefits such as forensics and management. As an experiment for this scenario, we measured how much detection coverage would be gained by adding the only truly free (GPL licensed) antivirus product, ClamAV, to an existing system employing only a single engine. Although ClamAV is not an especially effective engine by itself, it can add a significant amount of detection coverage, up to a 25% increase when paired with another engine as seen in Table 2.

## 7.5 Managing False Positives

The use of parallel detection engines has important implications for the management of false positives. While multiple detection engines can increase detection coverage, the number of false positives encountered during normal operation may increase when compared to a single engine. While antivirus vendors try hard to reduce false positives, they can severely impair productivity and take weeks to be corrected by a vendor.

The proposed architecture provides the ability to aggregate results from different detection techniques which enables the unique ability to trade-off detection coverage for false positive resistance. If an administrator wanted maximal detection coverage they could set the aggregation function to declare a candidate file unsafe if *any* detector indicated the file malicious. However, a false positive in any of the detector would cause the aggregator to declare the file unsafe.

In contrast, an administrator more concerned about false positives may set the aggregation function to declare a candidate file unsafe if at least half of the detectors deemed the file malicious. In this way multiple detection engines can be used to reduce the impact of false positives associated with any single engine.

To explore this trade-off, we collected 12 real-world false positives that impact different detectors in CloudAV. These files range from printer drivers to password recovery utilities to self-extracting zip files. We defined a threshold, or confidence index, of the number of engines required to detect a file before deeming it unsafe. For each threshold value, we measured the number of remaining false positives and also the corresponding detection rate of true positives.

The results of this experiment are seen in Table 3. At a threshold of 4 engines, all of the false positives are eliminated while only decreasing the overall detection coverage by less than 4%. As this threshold can be adjusted at any time via the management interface, it can set by an administrator based on the perceived threat model of the network and the actual number of false positives encountered during operation.

A second method of handling false positives is enabled by the centralized management of the network service. In the case of a standard host-based antivirus deployment, encountering a false positive may mean weeks of delay and loss of productivity while the antivirus vendor analyzes the false positive and releases an updated signature set to all affected clients. In the network-based architecture, the false positive can be added to a network-wide whitelist through the management interface in a matter of minutes by a local administrator. This whitelist management allows administrators to alleviate the pain of false positives and empowers them to cut out the antivirus vendor middle-man and make more informed and rapid decisions about threats on their network.

| Threshold | False Positives | Detection |
|---|---|---|
| 1 | 12 | 97.7% |
| 2 | 5 | 96.3% |
| 3 | 2 | 95.2% |
| 4 | 0 | 93.9% |

Table 3: The number of false positives observed at each engine threshold and the associated detection coverage over the full malware dataset.

## 7.6 Breaking Free of Vendor Lock-in

Finally, a serious issue associated with extensive deployments of host-based antivirus in a large enterprise or organizational network is vendor lock-in. Once a particular vendor has been selected through an organization's evaluation process and software is deployed to all departments, it is often hard to switch to a new vendor at a later point due to technical, management, and bureaucratic issues. In reality, organizations may wish to switch antivirus vendors for a number of reasons such as increased detection coverage, decreased licensing costs, or integration with network management devices.

The proposed antivirus architecture is innately vendor-neutral as it separates the acquisition of candidate files on the end host from the actual analysis and detection process performed in the network service. Therefore, even if only one detection engine is employed in the network service, a network administrator can easily replace it with another vendor's offering if so desired, without an upheaval of existing infrastructure.

## 8 Conclusion

To address the ever-growing sophistication and threat of modern malicious software, we have proposed a new model for antivirus deployment by providing antivirus functionality as a network service using N-version protection. This novel paradigm provides significant advantages over traditional host-based antivirus including better detection of malicious software, enhanced forensics capabilities, retrospective detection, and improved deployability and management. Using a production implementation and real-world deployment of the CloudAV platform, we evaluated the effectiveness of the proposed architecture and demonstrated how it provides significantly greater protection of end hosts against modern threats.

In the future, we plan to investigate the application of N-version protection to intrusion detection, phishing, and other realms of security that may benefit from heterogeneity. We also plan to open our backend analysis infrastructure to security researchers to aid in the detection and classification of collected malware samples.

## Acknowledgments

This work was supported in part by the Department of Homeland Security (DHS) under contract number NBCHC060090 and by the National Science Foundation (NSF) under contract number CNS 0627445.

## References

[1] Adobe Systems Incorporated. Apsb07-18: Adobe reader and acrobat vulnerability. http://www.adobe.com/support/security/bulletins/apsb07-18.html, 2007.

[2] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 1985.

[3] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The nepenthes platform: An efficient approach to collect malware. In *9th International Symposium On Recent Advances In Intrusion Detection*. Springer-Verlag, 2006.

[4] Josh Ballard. An Eye on the Storm: Inside the Storm Epidemic. 41st Meeting of the North Americian Network Operators Group, October 2007.

[5] Barracuda Networks. Barracuda spam firewall. http://www.barracudanetworks.com, 2007.

[6] Carsten Willems and Thorsten Holz. Cwsandbox. http://www.cwsandbox.org/, 2007.

[7] Cloudmark. Cloudmark authority anti-virus. http://www.cloudmark.com, 2007.

[8] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementaiton (OSDI)*, December 2002.

[9] F-Secure Corporation. F-secure mobile anti-virus. http://mobile.f-secure.com/, 2007.

[10] Gartner, Inc. Forecast: Security software worldwide, 2006-2011, update. http://www.gartner.com/DisplayDocument?ref=g_search&id=510567&subref=ad%vsearch, 2007.

[11] Hispasec Sistemas. Virus total. http://virustotal.com, 2004.

[12] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC 1997)*, May 1997.

[13] Kaspersky Lab. Kaspersky mobile security. http://usa.kaspersky.com/products_services/antivirus-mobile.php, 2007.

[14] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, 2003.

[15] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. *Proceedings of the USENIX/ACM Internet Measurement Conference*, October 2005.

[16] Mathias Rauen. madcodehook. http://madshi.net/, 2008.

[17] Microsoft. Microsoft security intelligence report: July-december 2006. http://www.microsoft.com/technet/security/default.mspx, May 2007.

[18] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of the 16th USENIX Security Symposium*, August 2007.

[19] Lajos Nagy, Richard Ford, and William Allen. N-version programming for the detection of zero-day exploits. In *IEEE Topical Conference on Cybersecurity*, Daytona Beach, Florida, USA, 2006.

[20] Arbor Networks. Arbor malware library (AML). http://www.arbornetworks.com, 2007.

[21] NIST/DHS/US-CERT. National vulnerability database. http://nvd.nist.gov/, 2007.

[22] Norman Solutions. Norman sandbox whitepaper. http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf, 2003.

[23] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Rethinking antivirus: Executable analysis in the network cloud. In *2nd USENIX Workshop on Hot Topics in Security (HotSec 2007)*, August 2007.

[24] John Ogness. Dazuko: An open solution to facilitate on-access scanning. *Virus Bulletin*, 2003.

[25] Niels Provos. Spybye. http://www.monkey.org/~provos/spybye, 2007.

[26] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001.

[27] SANS. Top-20 2007 security risks. http://www.sans.org/top20/, 2007.

[28] S. Sidiroglou, J. Ioannidis, A.D. Keromytis, and S.J. Stolfo. An Email Worm Vaccine Architecture. *Proceedings of the 1st Information Security Practice and Experience Conference (ISPEC)*, pages 97–108, 2005.

[29] Stelios Sidiroglou, Angelos Stavrou, and Angelos D. Keromytis. Mediated overlay services (moses): Network security as a composable service. In *Proceedings of the IEEE Sarnoff Symposium*, Princeton, NJ, USA, 2007.

[30] Symantec Corporation. Symantec security advisory (sym06-010). http://www.symantec.com/avcenter/security/Content/2006.05.25.html, 2006.

[31] Symantec Corporation. Symantec mobile antivirus for windows mobile. http://www.symantec.com/norton/products/overview.jsp?pcid=pf&pvid=smavw%m, 2007.

[32] Symantec Security Response Team. Ms word exploit creation tool. http://www.symantec.com/enterprise/security_response/weblog/2007/04/ms_%word_exploit_creation_tool.html, 2007.

[33] Vinod Yegneswaran, Paul Barford, and Somesh Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of Network and Distributed System Security Symposium (NDSS '04)*, San Diego, CA, February 2004.

# Highly Predictive Blacklisting

Jian Zhang
*SRI International*
*Menlo Park, CA 94025*

Phillip Porras
*SRI International*
*Menlo Park, CA 94025*

Johannes Ullrich
*SANS Institute*
*Bethesda, MD 20814*

## Abstract

The notion of blacklisting communication sources has been a well-established defensive measure since the origins of the Internet community. In particular, the practice of compiling and sharing lists of the worst offenders of unwanted traffic is a blacklisting strategy that has remained virtually unquestioned over many years. But do the individuals who incorporate such blacklists into their perimeter defenses benefit from the blacklisting contents as much as they could from other list-generation strategies? In this paper, we will argue that there exist better alternative blacklist generation strategies that can produce higher-quality results for an individual network. In particular, we introduce a blacklisting system based on a relevance ranking scheme borrowed from the link-analysis community. The system produces customized blacklists for individuals who choose to contribute data to a centralized log-sharing infrastructure. The ranking scheme measures how closely related an attack source is to a contributor, using that attacker's history and the contributor's recent log production patterns. The blacklisting system also integrates substantive log prefiltering and a severity metric that captures the degree to which an attacker's alert patterns match those of common malware-propagation behavior. Our intent is to yield individualized blacklists that not only produce significantly higher hit rates, but that also incorporate source addresses that pose the greatest potential threat. We tested our scheme on a corpus of over 700 million log entries produced from the DShield data center and the result shows that our blacklists not only enhance hit counts but also can proactively incorporate attacker addresses in a timely fashion. An early form of our system have been fielded to DShield contributors over the last year.

## 1 Introduction

A network address blacklist represents a collection of source IP addresses that have been deemed undesirable, where typically these addresses have been involved in some previous illicit activities. For example, DShield (a large-scale security-log sharing system) regularly compiles and posts a firewall-parsable blacklist of the most prolific attack sources seen by its contributors [17]. With more than 1700 contributing sources providing a daily stream of 30 million security log entries, such daily blacklists provide an informative view of those class C subnets that are among the bane of the Internet with respect to unwanted traffic. We refer to the blacklists that are formulated by a large-scale alert repository and consist of the most prolific sources in the repository's collection of data as the *global worst offender list* (GWOL). Another strategy for formulating network address blacklists is for an individual network to create a local blacklist based entirely on its own history of incoming communications. Such lists are often culled from a network's private firewall log or local IDS alert store, and incorporate the most repetitive addresses that appear within the logs. We call this blacklist scheme the *local worst offender list* (LWOL) method.

The GWOL and LWOL strategies have both strengths and inherent weaknesses. For example, while GWOLs provide networks with important information about highly prolific attack sources, they also have the potential to exhaust the subscribers' firewall filter sets with addresses that will simply never be encountered. Among the sources that do target the subscriber, GWOLs may miss a significant number of attacks, in particular when the attack sources prefer to choose their targets more strategically, focusing on a few known vulnerable networks [4]. Such attackers are not necessarily very prolific and are hence elusive to GWOLs. The sources on an LWOL have repetitively sent unwanted communications to the local network and are likely to continue doing so. However, LWOLs are limited by being entirely reactive – they only capture attackers that have been pounding the local network and hence cannot provide a potential for the blacklist consumer to learn of attack sources before these sources reach their networks.

Furthermore, both types of lists suffer from the fact that an attack source does not achieve candidacy until it has produced a sufficient mass of communications. That is, although it is desirable for firewall filters to include an attacker's address *before* it has saturated the network, neither GWOL nor LWOL offer a solution that can provide such timely filters. This is a problem particularly with GWOL. Even after an attacker has produced significant illicit traffic, it may not show up as a prolific source within the security log repository, because the data contributors of the repository are a very small set of networks on the Internet. Even repositories such as DShield that receive nearly 1 billion log entries per month represent only a small sampling of Internet activity. Significant attacker sources may elude incorporation into a blacklist until they have achieved extensive saturation across the Internet.

In summary, a high-quality blacklist that fortifies network firewalls should achieve high hit rate, should incorporate addresses in a timely fashion, and should proactively include addresses even when they have not been encountered previously by the blacklist consumer's network. Toward this goal, we present a new blacklist generation system which we refer to as the highly predictive blacklisting (HPB) system. The system incorporates 1) an automated log prefiltering phase to remove unreliable alert contents, 2) a novel relevance-based attack source ranking phase in which attack sources are prioritized on a per-contributor basis, and 3) a severity analysis phase in which attacker priorities are adjusted to favor attackers whose alerts mirror known malware propagation patterns. The system constructs final individualized blacklists for each DShield contributor by a weighted fusion of the relevance and severity scores.

HPB's underlying relevance-based ranking scheme represents a significant departure from the long-standing LWOL and GWOL strategies. Specifically, the HPB scheme examines not just *how many* targets a source address has attacked, but also *which* targets it has attacked. In the relevance-based ranking phase, each source address is ranked according to how closely related the source is to the target blacklist subscriber. This relevance measure is based on the attack source similarity patterns that are computed across all members of the DShield contributor pool (i.e., the amount of attacker overlap observed between the contributors). Using a data correlation strategy similar to hyper-text link analysis, such as Google's PageRank [2], the relationships among all the contributors are iteratively explored to compute an individual relevance value from each attacker to each contributor.

We evaluated our HPB system using more than 720 million log entries produced by DShield contributors from October to November 2007. We contrast the performance of the system with that of the corresponding GWOLs and LWOLs, using identical time windows, input data, and blacklist lengths. Our results show that for most contributors (more than 80%), our blacklist entries exhibit significantly higher hit counts over a multiday testing window than both GWOL and LWOL. Further experiments show that our scheme can proactively incorporate attacker addresses into the blacklist before these addresses reach the blacklist consumer network, and it can do so in a timely fashion. Finally, our experiments demonstrate that the hit count increase is consistent over time, and the advantages of our blacklist remain stable across various list lengths and testing windows.

The contribution of this paper is the introduction of the highly predictive blacklisting system, which includes our methodology for prefiltering, relevance-based ranking, attacker severity ranking, and final blacklist construction. Ours is the first exploration of a link-analysis-based scheme in the context of security filter production and to quantify the predictive quality of the resulting data. The HPB system is also one of the only new approaches we are aware of for large-scale blacklist publication that has been proposed in many years. However, our HPB system is applicable only to those users who participate as active contributors to collaborative security log data centers. Rather than a detriment, we hope that this fact provides some operators a tangible incentive to participate in security log contributor pools. Finally, the system discussed in this paper, while still a research prototype, has been fully implemented and deployed for nearly a year as a free service on the Internet at DShield.org. Our experience to date leads us to believe that this approach is both scalable and feasible for daily use.

The rest of the paper is organized as follows. Section 2 provides a background on previous work in blacklist generation and related topics. In Section 3 we provide a detailed description of the Highly Predictive Blacklist system. In Section 4 we present a performance evaluation of HPBs, GWOLs, and LWOLS, including assessments of the extent to which the above three desired blacklist properties (hit rate, proactive appearance, and timely inclusion) are realized by these three blacklists. In Section 5 we present a prototype implementation of the HPB system that is freely available to DShield.org log contributors, and we summarize our key findings in Section 6.

## 2 Related Work

Network address and email blacklists have been around since the early development of the Internet [6]. Today, sites such as DShield regularly compile and publish firewall-parsable filters of the most prolific attack sources reported to its website [17]. DShield represents a centralized approach to blacklist formulation, providing a daily perspective of the malicious background radiation that plagues the Internet [15, 20]. Other recent examples of computer and network blacklists include IP and DNS blacklists to help networks detect and block unwanted web content, SPAM producers, and phishing sites, to name a few [7, 8, 17, 18]. The HPB system presented here complements, but does not displace these resources or their blacklisting strategies. In addition, HPBs are only applicable to active log contributors (we hope as an incentive), not as generically publishable one-size-fits-all resources.

More agile forms of network blacklisting have also been explored, with the intention of rapidly publishing perimeter filters to control actively spreading malware epidemics [1, 3, 12, 14]. For example, in [14] a peer-to-peer blacklisting scheme is proposed, where each network incorporates an address into its local blacklist when a threshold number of peers have reported attacks from this address. We separate our HPB system from these malware defense schemes. While the HPB system does incorporate a malware-oriented attacker severity metric into its final blacklist selection, we have not contemplated nor propose HPBs for use in the context of dynamic quarantine defenses for malware epidemics.

One key insight that inspired the HPB relevance-based ranking scheme was raised by Katti et al. [10], who identified the existence of stable correlations among the attackers reported by security log contributors. Here we introduce a relevance-based recommendation scheme that selects candidate attack sources based on the attacker overlaps found among peer contributors. This relevance-based ranking scheme can be viewed as a random walk on the correlation graph, going from one node to another following the edges in the graph with the probability proportional to the weight of the graph. This form of random walk has been applied in link-analysis systems such as Google's PageRank [2], where it is used to estimate the probability that a webpage may be visited. Similar link analysis has been used to rank movies [13] and reading lists [19].

The problem of predicting attackers has also been recently considered in [24] using a Guassian process model. However, [24] purely focused on developing statistical learning techniques for attacker prediction based on collaborative filtering. In this paper, we present a comprehensive blacklisting generation system that considers many other characteristics of attackers. The prediction part is only one component in our system. Furthermore, the prediction model presented here is completely different from the one in [24] (Gaussian process model in [24] and link analysis model here). By taking some penalty in predictive power, the prediction model presented here is much more scalable, which is of necessity for implementing a deployable service (Section 5).

Finally, [23] provides a six-page summary of the earliest release of our DShield HPB service, including a high-level description of an early ranking scheme. In this paper we have substantially expanded this algorithm and present its full description for the first time. This present paper also introduces the integration of metrics to capture attack source maliciousness in its final rank selection, and presents the full blacklist construction system. We also present our quantitative evaluation of multiple system properties, and address several open questions that have been raised over the past year since our initial prototype.

## 3 Blacklisting System

We illustrate our blacklisting system in **Figure 1**. The system constructs blacklists in three stages. First, the security alerts supplied by sensors across the Internet are preprocessed. This removes known noises in the alert collection. We call this the *prefiltering* stage. The preprocessed data are then fed into two parallel engines. One ranks, for each contributors, the attack sources according to their relevance to that contributor. The other scores the sources using a severity assessment that measures their maliciousness. The relevance ranking and the severity score are combined at the last stage to generate a final blacklist for each contributor.

We descibe the prefiltering process in Section 3.1, relevance ranking in Section 3.2, severity score in Section 3.3 and the final production of the blacklists in Section 3.4.

### 3.1 Prefiltering Logs for Noise Reduction

One challenge to producing high-quality threat intelligence for use in perimeter filtering is that of reducing the amount of *noise* and erroneous data that may exist in the input data that drives our blacklist construction algorithm. That is, in addition to the unwanted port scans, sweeps, and intrusion attempts reported daily within the DShield log data, there are also commonly produced log entries that arise from nonhostile activity, or activity from which useful filters cannot be reliably derived. While it is not possible to separate attack from nonattack data, the HPB system prefilters from consideration logs that match criteria that we have been able to empirically identify as commonly occurring nonuseful input for blacklist construction purposes.

As a preliminary step prior to blacklist construction, we apply three filtering techniques to the DShield alert logs. First, the HPB system removes from consideration DShield logs produced from attack sources from invalid or unassigned IP address space. Here we employ the

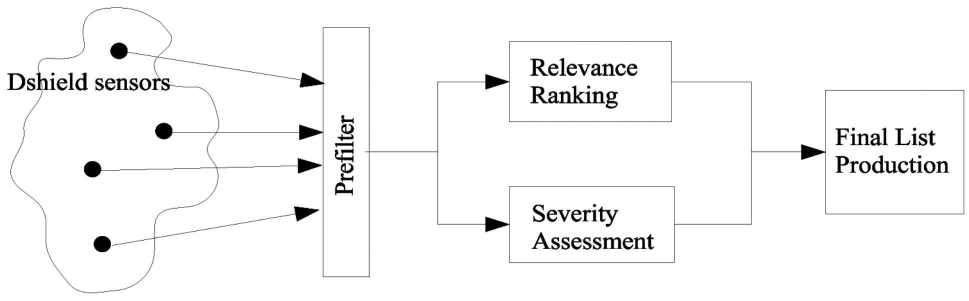


Figure 1: Blacklisting system architecture

bogon list created by the Cymru team that captures addresses that are reserved, not yet allocated, or delegated by the Internet Assigned Number Authority [16]. Typically, such addresses should not be routed, but otherwise do appear anyway in the DShield data. In addition, reserved addresses such as the 10.x.x.x or 192.168.x.x may also appear in misconfigured contributor logs that are not useful for translating into blacklists.

Second, the system prefilters from consideration network addresses from Internet measurement services, web crawlers, or common software update sources. From experience, we have developed a whitelist of highly common sources that, while innocuous from an intrusion perspective, often generate alarms in DShield contributor logs.

Finally, the HPB system applies heuristics to avoid common false positives that arise from commonly timed-out network services. Specifically, we exclude logs produced from source ports TCP 53 (DNS), 25 (SMTP), 80 (HTTP), and 443 (often used for secure web, IMAP, and VPN), and from destination ports TCP 53 (DNS) and 25 (SMTP). Firewalls will commonly time out sessions from these services when the server or client becomes unresponsive or is slow. In practice, the combination of these prefiltering steps provides approximately a 10% reduction in the DShield input stream prior delivery to the blacklist generation system.

## 3.2 Relevance Ranking

Our notion of attacker relevance is a measure that indicates how close the attacker is related to a particular blacklist consumer. It also reflects the likelihood to which the attacker may come to the blacklist consumer in the near future. Note that this relevance is orthogonal to metrics that measure the severity (or benignness) of the source, which we will discuss in the next section.

In our context, the blacklist consumers are the contributors that supply security logs to a log-sharing repository such as DShield. Recent research has observed the existence of attacker overlap correlations between DShield contributors [10], i.e., there are pairs of contributors that share quite a few common attackers, where the common attacker is defined as a source address that both contributors have logged and reported to the repository. This research also found that this attacker overlap phenomenon is not due to attacks that select targets randomly (as in a random scan case). The correlations are long lived and some of them are independent of address proximity. We exploit these overlap relationships to measure attacker relevance.

We first illustrate a simple concept of attacker relevance. Consider a collection of security logs displayed in a tabular form as shown in **Table 1**. We use the rows of the table to represent attack sources and the columns to represent contributors. We refer to the unique source addresses that are reported within the log repository as *attackers*, and use the terms "attacker" and "source" interchangeably. Since the contributors are also the targets of the logged attacks, we refer to them as *victims*. We will use the terms "contributor" and "victim" interchangeably. An asterisk "*" in the table cell indicates that the corresponding source has reportedly attacked the corresponding contributor.

|  | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| $s_1$ | * | * |  |  |  |
| $s_2$ | * | * |  |  |  |
| $s_3$ | * |  | * |  |  |
| $s_4$ |  | * | * |  |  |
| $s_5$ |  | * |  |  |  |
| $s_6$ |  |  |  | * | * |
| $s_7$ |  |  | * |  |  |
| $s_8$ |  |  | * | * |  |

Table 1: Sample Attack Table

Let us assume that **Table 1** represents a series of logs contributed in the recent past by our five victims, $v_1$ through $v_5$. Now suppose we would like to calculate the relevance of the sources for contributor $v_1$ based on these attack patterns. From the attack table we observe that contributors $v_1$ and $v_2$ share multiple common attackers. $v_1$ also shares one common attack source ($s_3$) with $v_3$, but does not share attacker overlap with the other contributors. Given this observation, between sources $s_5$ and $s_6$, we would say that $s_5$ has more relevance to $v_1$ than $s_6$ because $s_5$ has reportedly attacked $v_2$, which has recently experienced multiple attack source overlaps with $v_1$. But the victims of $s_6$'s attacks share no overlap with $v_1$. Note that this relevance measure is quite different from the measures based on how prolific the attack source has been. The latter would favor $s_6$ over $s_5$, as $s_6$ has attacked more victims than $s_5$. In this sense, *which* contributors a source has attacked is of greater significance to our scheme than how many victims it has attacked. Similarly, between $s_5$ and $s_7$, $s_5$ is more relevant, because the victim of $s_5$ ($v_2$) shares more common attacks with $v_1$ than the victim of $s_7$ ($v_3$). Finally, because $s_4$ has attacked both $v_2$ and $v_3$, we would like to say that it is the most relevant among $s_4, s_5, s_6$, and $s_7$.

To formalize the above intuition, we model the attack correlation relationship between contributors using a *correlation graph*, which is a weighted undirected graph $G = (V, E)$. The nodes in the graph consist of the contributors $V = \{v_1, v_2, \ldots\}$. There is an edge between node $v_i$ and $v_j$ if $v_i$ is correlated with $v_j$. The weight on the edge is determined by the strength of the correlation (i.e., occurrences of attacker overlap) between the two corresponding contributors. We now introduce some notation for the relevance model.

Let $n$ be the number of nodes (number of contributors) in the correlation graph. We use $\mathbf{W}$ to denote the adjacency matrix of the correlation graph, where the entry $\mathbf{W}_{(i,j)}$ in this matrix is the weight of the edge between node $v_j$ and $v_i$. For a source $s$, we denote by $T(s)$ the set of contributors that have reported an attack from $s$. $T(s)$ can be written in a vector form $\mathbf{b}^s = \{b^s_1, b^s_2, \ldots, b^s_n\}$ such that $b^s_i = 1$ if $v_i \in T(s)$ and $b^s_i = 0$ otherwise. We also associate with each source $s$ a relevance vector $\mathbf{r}^s = \{r^s_1, r^s_2, \ldots, r^s_n\}$ such that $r^s_v$ is the relevance value of attacker $s$ with respect to contributor $v$. We use lowercase boldface to indicate vectors and uppercase boldface to indicate matrices. **Table 2** summarizes our notation.

We now describe how to derive the matrix $\mathbf{W}$ from the attack reports. Consider the following two cases. In Case 1, contributor $v_i$ sees attacks from 500 sources and $v_j$ sees 10 sources. Five of these sources attack both $v_i$ and $v_j$. In Case 2, there are also five common sources. However, $v_i$ sees only 50 sources and $v_j$ sees 10. Although the number of overlapping sources is the same (i.e., 5 common sources), the strength of connection between $v_i$ and $v_j$ is different in the two cases. If a contributor observes a lot of attacks, it is expected that there should be more overlap between this contributor and the others. Let $m_i$ be the number of sources seen by $v_i$, $m_j$ the number seen by $v_j$, and $m_{ij}$ the number of common attack sources. The ratio $\frac{m_{ij}}{m_i}$ shows how important $v_i$ is for $v_j$ while $\frac{m_{ij}}{m_j}$ shows how important $v_j$ is for $v_i$. Since we want $\mathbf{W}_{(i,j)}$ to reflect the strength of the connection between $v_i$ and $v_j$, we set $\mathbf{W}_{(i,j)} = \frac{m_{ij}}{m_i} \cdot \frac{m_{ij}}{m_j}$. One may view this new $\mathbf{W}$ as a standardized correlation matrix. **Figure 2** shows the matrix $\mathbf{W}$ for **Table 1** constructed using this method.

| | |
|---|---|
| $n$ | # of contributors |
| $v_i$ | $i$-th contributor |
| $\mathbf{W}$ | Adjacency matrix of the correlation graph |
| $T(s)$ | Set of contributors that have reported attack(s) from source $s$ |
| $\mathbf{b}^s$ | Attack vector for source $s$. $b^s_i = 1$ if $v_i \in T(s)$ and 0 otherwise |
| $\mathbf{r}^s$ | Relevance vector for source $s$. $r^s_v$ is the relevance value of attacker $s$ with respect to contributor $v$ |

Table 2: Summary of Relevance Model Notations

$$\begin{pmatrix} 0 & 0.33 & 0.083 & 0 & 0 \\ 0.33 & 0 & 0.063 & 0 & 0 \\ 0.083 & 0.063 & 0 & 0.13 & 0 \\ 0 & 0 & 0.13 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0 \end{pmatrix}$$

Figure 2: Standardized Correlation Matrix for Attack Table 1

Given this correlation matrix, we follow the aforementioned intuition and calculate the relevance as $r^s_i = \sum_{j \in T(s)} \mathbf{W}_{(i,j)}$. This is to say that if the repository reports that source $s$ has attacked contributor $v_j$, this fact contributes a value of $\mathbf{W}_{(i,j)}$ to the source's relevance with respect to the victim $v_i$. Written in vector form, it gives us

$$\mathbf{r}^s = \mathbf{W} \cdot \mathbf{b}^s. \tag{1}$$

The above simple relevance calculation lacks certain desired properties. For example, the simple relevance value is calculated solely from the observed activities from the source by the repository contributors. In some cases, this observation does not represent the complete view of the source's activity. One reason is that the contributors consist of only a very small set of networks in the Internet. Before an attacker saturates the Internet with malicious activity, it is often the case that only a few contributors have observed the attacker. The attacker may be at its early stage or it has attacked many places, most of which do not participate in the security log sharing system. Therefore, one may want a relevance measure that has a "look-ahead" capability. That is, the relevance calculation should take into consideration possible future observations of the source and include these anticipated observations from the contributors into the relevance values.

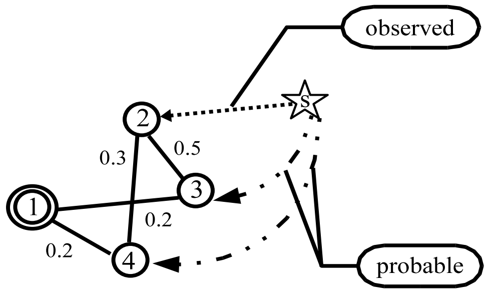


Figure 3: Relevance Evaluation Considers Possible Future Attacks

**Figure 3** gives an example where one may apply this "look-ahead" feature. (Examples here are independent of the one shown in **Table 1**.) The correlation graph of **Figure 3** consists of four contributors numbered 1, 2, 3, and 4. Contributor 2 reported an attack from source $s$ (represented by the star). Our goal is to evaluate how relevant this attacker is to contributor 1 (double-circled node). Using **Equation 1**, the relevance would be zero. However, we observe that $s$ has relevance 0.5 with respect to contributor 3 and relevance 0.3 with respect to contributor 4. Although at this time, contributors 3 and 4 have not observed $s$ yet, there may be possible future attacks from $s$. In anticipation of this, when evaluating $s$'s relevance with respect to contributor 1, contributors 3 and 4 pass to contributor 1 their relevance values after multiplying them with the weights on their edges, respectively. The attacker's relevance value for contributor 1 then is 0.5*0.2+0.3*0.2 = 0.16. Note that, had $s$ actually attacked contributors 3 and 4, the contributors would have passed the relevance value 1 (again after multiplying them with the weights on the edges) to contributor 1.

This can be viewed as a relevance propagation process. If a contributor $v_i$ observed an attacker, we say that the attacker has an initial relevance value 1 for that contributor. Following the edges that go out of the contributor, a fraction of this relevance can be distributed to the neighbors of the contributor in the graph. Each of $v_i$'s neighbors receives a share of relevance that is proportional to the weight on the edge that connects the neighbor to $v_i$. Suppose $v_j$ is one of the neighbors. A fraction of the relevance received by $v_j$ is then further distributed, in similar fashion, to its neighbors. The propagation of relevance continues until the relevance values for each contributor reach a stable state.

This relevance propagation process has another benefit besides the "look-ahead" feature. Consider the correlation graph given in **Figure 4** (a). The subgraph formed by nodes 1, 2, 3, and 4 is very different from that formed by nodes 1, 5, 6, and 7. The subgraph from nodes 1, 2, 3, and 4 is well connected (in fact it forms a clique). The contributors in the subgraph are thus more tied together. We call them a *correlated group*. (We use a dotted circle to indicate the correlated group in **Figure 4**.) There may be certain intrinsic similarities between the members in the correlated group (e.g., IP address proximity, similar vulnerability). Therefore, it is natural to assign more relevance to source addresses that have attacked other contributors in the same correlated group. For example, consider the sources $s$ and $s'$ in **Figure 4**. They both attacked three contributors. All the edges in the correlation graph have the same weights. (Hence, we omitted the weights in the figure.) We would like to say that $s$ is more relevant than $s'$ for contributor 1. If we calculate the relevance value by **Equation 1**, the values would be the same for the two attackers. Relevance propagation helps to give more value to the attacker $s$ because members of the correlated group are well connected. There are more paths in the subgraph that lead from the contributors where the attack happened to the contributor for which we are evaluating the attacker relevance. For example, the relevance from contributor 2 can propagate to contributor 3 and then to contributor 1. It can also go to contributor 4 and then to contributor 1. This is effectively the same as having an edge with larger weight between the contributors 2 and 1. Therefore, relevance propagation can effectively discover and adapt to the structures in the correlation graph. The relevance values assigned then reflect certain intrinsic relationships among contributors.

We extend **Equation 1** to employ relevance propagation. If we propagate the relevance values to the immediate neighbors in the correlation graph, we obtain a relevance vector $\mathbf{W} \cdot \mathbf{b^s}$ that represents the propagated values. Now we propagate the relevance values one more hop. This gives us $\mathbf{W} \cdot \mathbf{W} \cdot \mathbf{b^s} = \mathbf{W}^2 \cdot \mathbf{b^s}$. The relevance vector that reflects the total relevance value each contributor receives is then $\mathbf{W} \cdot \mathbf{b^s} + \mathbf{W}^2 \cdot \mathbf{b^s}$. If we let the propagation process iterate indefinitely, the relevance vector would become $\sum_{i=1}^{\infty} \mathbf{W}^i \cdot \mathbf{b^s}$. There is a technical detail in this process we need to resolve. Naturally, we would like the relevance value to decay along the path of propagation. The further it goes on the graph, the smaller its contribution becomes. To achieve this, we scale the matrix $\mathbf{W}$ by a constant $0 < \alpha < 1$ such that the 2-norm of the new matrix $\alpha\mathbf{W}$ becomes smaller than one. With this modification, an attacker will have

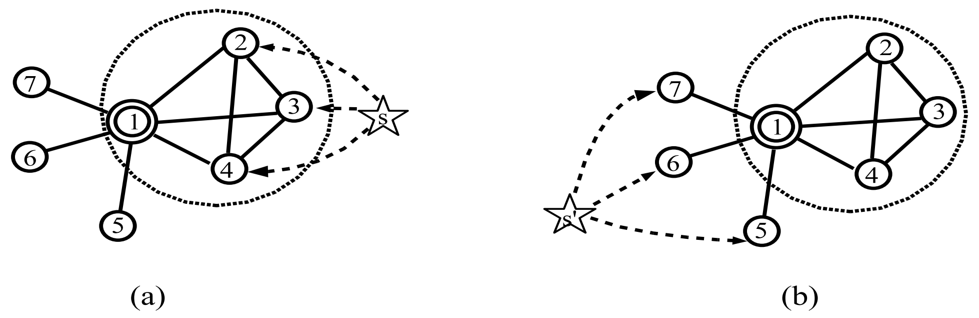


Figure 4: Attacks on Members in a Correlated Group Contribute More Relevance

only a negligible relevance value to contributors that are far away in the correlation graph. Putting the above together, we compute the relevance vector by the following equation:

$$\mathbf{r}^s = \sum_{i=1}^{\infty} (\alpha \mathbf{W})^i \cdot \mathbf{b^s} \tag{2}$$

We observe that $\mathbf{b^s} + \mathbf{r}^s$ is the solution for $\mathbf{x}$ in the following system of linear equations:

$$\mathbf{x} = \mathbf{b^s} + \alpha \mathbf{W} \cdot \mathbf{x} \tag{3}$$

The linear system described by **Equation 3** is exactly the system used by Google's PageRank [2]. PageRank analyzes the link structures of webpages to determine the relevance of each webpage with respect to a keyword query. In PageRank, $b^s$ is set to be an all-one vector and $\mathbf{W}$ is determined by letting $\mathbf{W}_{(i,j)}$ be 1/(# of outgoing links on page $j$) if one of these outgoing links points to webpage $i$, and $\mathbf{W}_{(i,j)} = 0$ otherwise. Therefore, PageRank propagates relevance where every node provides an initial relevance value of one. In our relevance calculation, only nodes whose corresponding contributors have reported the attacker are assigned one unit of initial relevance. Similar to the PageRank values that reflect the link structures of the webpages, our relevance values reflect the structure of the correlation graph that captures intrinsic relationships among the contributors.

**Equation 3** can be solved to give $\mathbf{x} = (\mathbf{I} - \alpha \mathbf{W})^{-1} \cdot \mathbf{b}^s$, where $\mathbf{I}$ is the identity matrix. Also, since $\mathbf{x} = \mathbf{r}^s + \mathbf{b}^s$, $\mathbf{r}^s = (\mathbf{I} - \alpha \mathbf{W})^{-1} \cdot \mathbf{b}^s - \mathbf{b}^s = [(\mathbf{I} - \alpha \mathbf{W})^{-1} - \mathbf{I}] \cdot \mathbf{b}^s$. This gives the relevance vector for each attack source. The sources are then ranked, for each contributor, according to the relevance values. As each attack source has a potentially different relevance value for each contributor, the rank of a source with respect to different contributors is different. Note that our concept of relevance measure and relevance propagation does not depend on a particular choice of the $\mathbf{W}$ matrix. As long as $\mathbf{W}$ reflects the connection weight between the contributors, our relevance measure applies.

## 3.3 Analyzing Attack Pattern Severity

We now consider the problem of measuring the degree to which each attack source exhibits known patterns of malicious behavior. In the next section, we will disuss how this measure can be fused into our final blacklist construction decisions. In this section we will describe our model of malicious behavior and the attributes we extract to map each attacker's log production patterns to this model.

Our model of *malicious behavior*, in this instance, focuses on identifying typical scan-and-infect malicious software (or malware). We define our malware behavior pattern as that of an attacker who conducts an IP sweep to small sets of ports that are known to be associated with malware propagation or backdoor access. This behavior pattern matches the malware behavior pattern documented by Yegeneswaren et.al. in [20], as well as our own most recent experiences (within the last twelve months) of more than 20K live malware infections observed within our honeynet [21]. Other potential malware behavior patterns may be applied, for example, such as the scan-oriented malicious address detection schemes outlined in the context of dynamic signature generation [11] and malicious port scan analysis [9]. Regardless of the malware behavior model used, the design and integration of other severity metrics into the final blacklist generation process can be carried out in a similar fashion.

For the set of log entries over the relevance-calculation time window, we calculate several attributes for each attacker's /24 network address. (Our blacklists are specified on a per /24 basis, meaning that a single malicious address has the potential to induce a LAN-wide filter. This is standard practice for DShield and other blacklists.) For each attacker, we assign a score to target ports associated with the attacker, assigning a different weight depending on whether or not the port is associated with known malware communications.

Let $MP$ be the set of malware-associated ports, where we currently uses the definition in Figure 5. This $MP$

$$\begin{pmatrix} 53-UDP & 69-UDP & 137-UDP & 21-TCP & 53-TCP & 42-TCP \\ 135-TCP & 139-TCP & 445-TCP & 559-TCP & 1025-TCP & 1433-TCP \\ 2082-TCP & 2100-TCP & 2283-TCP & 2535-TCP & 2745-TCP & 2535-TCP \\ 3127-TCP & 3128-TCP & 3306-TCP & 3410-TCP & 5000-TCP & 5554-TCP \\ 6101-TCP & 6129-TCP & 8866-TCP & 9898-TCP & 10000-TCP & 10080-TCP \\ 12345-TCP & 11768-TCP & 15118-TCP & 17300-TCP & 27374-TCP & 65506-TCP \\ 4444-TCP & 9995-TCP & 9996-TCP & 17300-TCP & 3140-TCP & 9033-TCP \\ 1434-UDP & & & & & \end{pmatrix}$$

Figure 5: Malware Associated Ports

is derived from various AV lists and our honeynet experiences. We do not argue that this list is complete and can be expanded across the life of our HPB service. However, our experiences in live malware analysis indicate that the entries in $MP$ are both highly common and highly indicative of malware propagation.

Let the number of target ports that attacker $s$ connects to be $c_m$, and the total number of unique ports connected to be defined as $c_u$. We associate a weighting (or importance) factor $w_m$ for all ports in $MP$, and a weighting factor $w_u$ for all nonmalware ports. We then compute a malware port score ($PS$) metric for each attacker as follows:

$$PS(s) = \frac{(w_u \times c_u) + (w_m \times c_m)}{c_u} \quad (4)$$

Here, we intend $w_m$ to be of greater weight than $w_u$, and choose an initial default of $w_m = 4 * w_u$. $PS$ has the property that even if a large $c_m$ is found, if $c_u$ is also large (as in a horizontal portscan), then $PS$ will remain small. Again, our intention is to promote a malware behavior pattern in which malware propagation will tend to target fewer specific ports, and is not associated with attackers that engage in horizontal port sweeps.

Next, we compute the set of unique target IP addresses connected to by attacker $s$. We refer to this count as $TC(s)$. A large $TC$ represents confirmed IP sweep behavior, which we strongly associate with our malware behavior model. $TC$ is *the* exclusive prioritization metric used by GWOL, whereas here we consider $TC$ a secondary factor to $PS$ in computing a final malware behavior score. We could also include metrics regarding the number of DShield sensors (i.e., unique contributor IDs) that have reported the attacker, which arguably represents the degree of *consensus* in the contributor pool that the attack source is active across the Internet. However, the IP sweep pattern is of high interest, even when the IP sweep experiences may have been reported only by a smaller set of sensors.

Third, we compute an optional tertiary behavior metric that captures the ratio of national to international addresses that are targeted by attacker $s, IR(s)$. Within the DShield repository we find many cases of sources (such as from China, Russian, the Czech Republic) that exclusively target international victims. However, this may also illustrate a weakness in the DShield contributor pool, as there may be very few contributors that operate sensors within these countries. We incorporate a dampening factor $\delta$ ($0 \leq \delta \leq 1$) that allows the consumer to express the degree to which the $IR$ factor should be nullified in computing the final severity score for each attacker.

Finally, we compute a malware severity score $MS(s)$ for each candidate attacker that may appear in the set of final blacklist entries:

$$MS(s) = PS(s) + \log(TC(s)) + \delta \log(IR(s)) \quad (5)$$

The three factors are computed in order of significance in mapping to our malware behavior model. Logarithm is used because in our model, the secondary metric ($TC$) and the tertiary metric ($IR$) are less important than the malware port score and we only care about their order of magnitude.

## 3.4 Blacklist Production

For each attacker, we now have both its relevance ranking and its severity score. We can combine them to generate a final blacklist for each contributor.

For the final blacklist, we would like to include the attackers that have strong relevance and discard the non-relevant attackers. To generate a final list of length $L$, we use the attacker's relevance ranking to compile a candidate list of size $c \cdot L$. (We often set $c = 2$.) Then, we use severity scores of the attackers on the candidate list to adjust its ranking and pick the $L$ highest-ranked attackers to form the final list. Intuitively, the adjustment should promote the rank of an attacker if the severity assessment indicates that it is very malicious. Toward this goal, we define a final score that combines the attacker's relevance rank in the candidate list and its severity assessment. In particular, let $k$ be the relevance rank of the attacker $s$

(i.e., $s$ is the k-th entry in the candidate list). Recall from last section $MS(s)$ is the severity score of $s$. The final score $fin(s)$ is defined to be

$$fin(s) = k - \frac{L}{2} \cdot \Phi(MS(s)) \tag{6}$$

where

$$\Phi(x) = \frac{1}{2}(1 + erf(\frac{x - \mu}{d}))$$

where $erf(\cdot)$ is the "S" shaped Gaussian error function. We plot $\Phi(x)$ in **Figure 6** with $\mu = 4$ and different $d$.

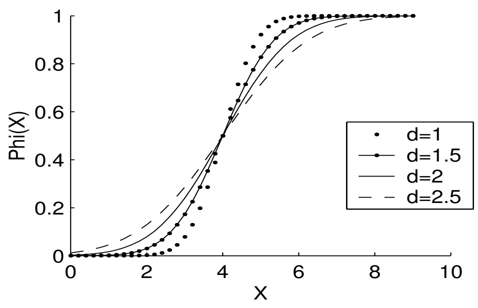


Figure 6: Phi with different d value

$\Phi(MS(s))$ promotes the rank of an attacker according to its maliciousness. The larger the value of $\Phi(MS(s))$ is, the more the attacker is moved above comparing to its original rank. A $\Phi(MS(s))$ of value 1 would then move the attacker above for one half of the size of the final list comparing to its original rank. The "S" shaped $\Phi(\cdot)$ transforms the severity assessment $MS(s)$ into a value between 0 and 1. The less-malicious attackers often give an assessment score below 3. After transformation, they will receive only small promotions. On the other hand, malicious attackers that give an assessment score above 7 will be highly promoted.

To generate the final list, we sort the $fin(s)$ values of the attackers in the candidate list and then pick $L$ of them that have the smallest $fin(s)$.

## 4 Experiment Results

We created an experimental HPB blacklist formulation system. To evaluate the HPBs, we performed a battery of experiments using the DShield.org security firewall and IDS log repository. We examined a collection of more than 720 million log entries produced by DShield contributors from October to November 2007. Since our relevance measure is based on correlations between contributors, HPB production is not applicable to contributors that have submitted very few reports (DShield has contributors that hand-select or sporadically contribute logs, providing very few alerts). We therefore exclude those contributors that we find effectively have no correlation with the wider contributor pool or simply have too few alerts to produce meaningful results. For this analysis, we found that we could compute correlation relationships for about 700 contributors, or 41% of the DShield contributor pool.

To assess the performance of the HPB system, we compare its performance relative to the standard DShield-produced GWOL [17]. In addition, we compare our HPB performance to that of LWOLs, which we compute individually for all contributors in our comparison set. For the purpose of our comparative assessment, we fixed the length of all three competing blacklists to exactly 1000 entries. However, after we present our comparative performance results, we will then continue our investigation by analyzing how the blacklist length affects the performance of the HPBs.

In the experiments, we generate GWOL, LWOL, and HPBs using data for a certain time period and then test the blacklists on data from the time window following this period. We call the period used for producing blacklists the *training window* and the period for testing the *prediction window*. In practice, the training period represents a snapshot of the most recent history of the repository, used to formulate each blacklist for a contributor that is then expected to use the blacklist for the length of the prediction window. The sizes of these two windows are not necessarily equal. We will first describe experiments that use 5-day lengths for both the training window and the prediction window. We then present experiments that investigate the effects of the two windows' lengths on HPB quality.

### 4.1 Hit Count Improvement

DShield logs submitted during the prediction window are used to determine how many sources included within a contributor's HPB are indeed encountered during that prediction window. We call this value the blacklist *hit count*. We view each blacklist address filter not encountered by the blacklist consumer as an *opportunity cost* to have prevented the deployment of other filters that could have otherwise blocked unwanted traffic. In this sense, we view our hit count metric as an important measure of the effectiveness of a blacklist formulation algorithm. Note that our HPBs are formulated with severity analysis while the other lists are not. As the severity analysis prefers malicious activities, we expect that the hits on the HPBs are more malicious.

To compare the three types of lists, we take 60 days of data, divided into twelve 5-day windows. We repeat the experiment 11 times using the $i$-th window as the training window and the $(i+1)$-th window as the testing window. In the training window, we construct HPB, LWOL, and

| Window | GWOL total hit | LWOL total hit | HPB total hit | HPB/GWOL | HPB/LWOL |
|---|---|---|---|---|---|
| 1 | 81937 | 85141 | 112009 | 1.36701 | 1.31557 |
| 2 | 83899 | 74206 | 115296 | 1.37422 | 1.55373 |
| 3 | 87098 | 96411 | 122256 | 1.40366 | 1.26807 |
| 4 | 80849 | 75127 | 115715 | 1.43125 | 1.54026 |
| 5 | 87271 | 88661 | 118078 | 1.353 | 1.33179 |
| 6 | 93488 | 73879 | 122041 | 1.30542 | 1.6519 |
| 7 | 100209 | 105374 | 133421 | 1.33143 | 1.26617 |
| 8 | 96541 | 91289 | 126436 | 1.30966 | 1.38501 |
| 9 | 94441 | 107717 | 128297 | 1.35849 | 1.19106 |
| 10 | 96702 | 94813 | 128753 | 1.33144 | 1.35797 |
| 11 | 97229 | 108137 | 131777 | 1.35533 | 1.21861 |
| Average | 90879 $\pm$ 6851 | 90978 $\pm$ 13002 | 123098 $\pm$ 7193 | 1.36 $\pm$ 0.04 | 1.37 $\pm$ 0.15 |

Table 3: Hit Number Comparison between HPB, LWOL and GWOL

| | Contributor Percentage | Average Increase | Median Increase | StdDev | Increase Range |
|---|---|---|---|---|---|
| Improved vs. GWOL | 90% | 51 | 22 | 89 | 1 to 732 |
| Poor vs. GWOL | 7% | -27 | -7 | 47 | -1 to -206 |
| Improved vs. LWOL | 95% | 75 | 36 | 90 | 1 to 491 |
| Poor vs. LWOL | 4% | -19 | -9 | 28 | -1 to -104 |

Table 4: Hit Count Performance, HPB vs. (GWOL and LWOL), Length 1000 Entries

GWOL. Then the three types of lists are tested on the data in the testing window.

Table 3 shows the total number of hits summed over the contributors for HPB, GWOL, and LWOL, respectively. It also shows the ratio of HPB hits over that of GWOL and LWOL. We see that in every window, HPB has more hits than GWOL and LWOL. Overall, HPBs predict 20-30% more hits than LWOL and GWOL. Note that there are quite large variances among the number of hits between time windows. Most of the variances, however, are not from our blacklist construction, rather they are from the variance among the number of attackers the networks experience in different testing windows.

| | Increase Average | Increase Median | Increase StdDev | Increase Range |
|---|---|---|---|---|
| vs. GWOL | 129 | 78 | 124 | 40 to 732 |
| vs. LWOL | 183 | 188 | 93 | 59 to 491 |

Table 5: Top 200 Contributors' Hit Count Increases (Blacklist Length 1000)

The results in Table 3 show HPB's hit improvement over time windows. We now investigate the distribution of the HPB's hit improvement across contributors in one time window. We use two quantities for comparison. The first is the hit count improvement, which is simply the HPB hit count minus the hit count of the other list. The second comparative measure we used is the relative hit count improvement (RI), which is the ratio in percentage of the HPB hit count increase over the other blacklist hit count. If the other list hit count is zero we define RI to be 100x the HPB hit count, and if both hit counts are zero we set RI to 100.

**Table 5** provides a summary of hit-count improvement for the 200 contributors where HPBs perform the best. The hit-count results for all the contributors are summarized in **Table 4**.

**Figure 7** compares HPB to GWOL. The left panel of the figure plots the histogram showing the distribution of the hit improvement across the contributors. The x-axis indicates improvements, and the hight of the bars represents the number of contributors whose improvement fall in the corresponding bin. Bars left to $x = 0$ represent contributors for whom the HPB has worse performance and bars on the right represent contributors for whom HPBs performed better. For most contributors, the improvment is positive. The largest improvement reaches 732. For only a few contributors, HPB performs worse in this time window.

The panel on the right of **Figure 7** plots the RI (ratio % of HPB's hit count increase over GWOL's hit count) distribution. We sort the RI values and plot them against the contributors. We label the x-axis by cummulative percentage, i.e., a tick on x-axis represents the percentage of contributors that lie to the left of the tick. For example, the tick 20 means 20 percent of the contributors lie left to this tick. There are contributors for which the RI value can be more than 3900. Instead of showing such large RI values, we cut off the plot at RI value 300. From the plot, we see that there are about 20% of contributors for which the HPBs achieve an RI more than 100, i.e., the HPB at least doubled the GWOL hit count. For about half of the contributors, the HPBs have about 25% more hits (an RI of 25). The HPBs have more hits than GWOL for almost 90% of the contributors. Only for a few con-

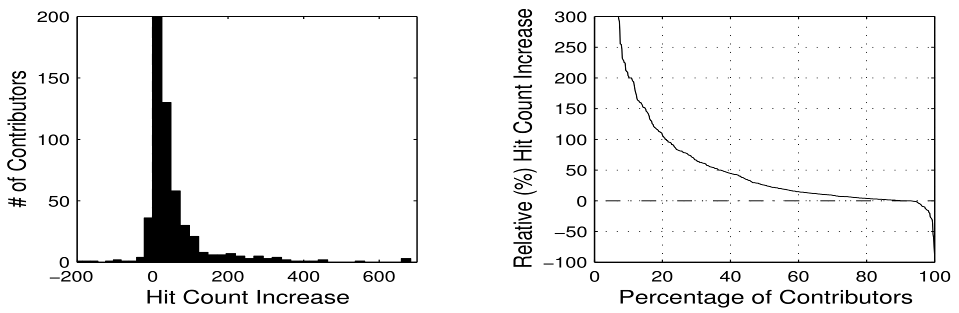


Figure 7: Hit Count Comparison of HPB and GWOL: Length 1000 Entries

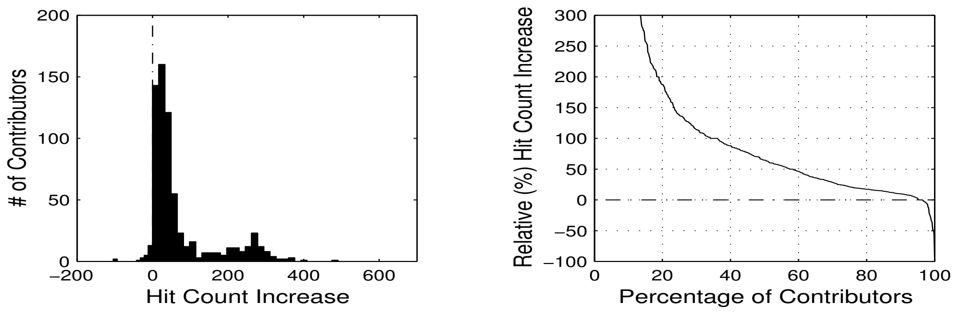


Figure 8: Hit Count Comparison of HPB and LWOL: Length 1000 Entries

tributors (about 7%), HPBs perform worse. (We discuss the reasons why HPB may perform worse in Section 4.4.)

**Figure 8** compares HPB hit counts to those of LWOL. The data are plotted in the same way as in **Figure 7**. Overall, HPBs demonstrate a performance advantage over LWOL. The IV and RI values also exhibit similar distributions. However, comparing **Figures 8** and **7**, we see that HPB has more hit improvement comparing to LWOL than to GWOL in this time window.

## 4.2 Prediction of New Attacks

One clear motivating assumption in secure collaborative defense strategies is that participants have the potential to prepare themselves from attacks that they have not yet encountered. We will say that a *new attack* occurs when a contributor produces a DShield log entry from a source that this contributor has never before reported. In this experiment, we show that HPB analysis provides contributors a potential to predict more new attacks than GWOL. (LWOL is not considered, since by definition it includes *only* attackers that are actively hitting the LWOL owner.) For each contributor, we construct two new HPB and GWOL lists with equal length of 1000 entries, such that no entries have been reported by the contributor during our training window. We call these lists HPB-local (HPB minus local) and GWOL-local (GWOL minus local), respectively. **Figure 9** compares HPB-local and GWOL-local on their ability to predict on new attack sources for the local contributor. These hit number plots demonstrate that HPB-local provides substantial improvement over the predictive value of GWOL.

## 4.3 Timely Inclusion of Sources

By timely inclusion, we refer to the ability of a blacklist to incorporate addresses relevant to the blacklist owner *before* those addresses have saturated the Internet. To investigate the timeliness of the GWOL, LWOL, and the HPB we examine how many contributors need to report a particular attacker before it can be included into the respective blacklists. We focus our attention on the set of attackers within these blacklists that *did* carry out attacks during the prediction window. And we use the number of distinct victims (contributors) that a source attacked in the training window to measure the extent to which the source has saturated the Internet. **Figure 10** plots the distribution of the number of distinct victims across different attackers on the three blacklists. As expected, the attackers that get selected on the GWOL were the most prolific in the training period. In particular, all the sources on the GWOL have attacked more than 20 contributors and almost 1/3 of them attacked more than 200 contributors. To some extent, these attackers have saturated the Internet with their activities. (DShield sensors are a very small sample of the Internet. A random at-

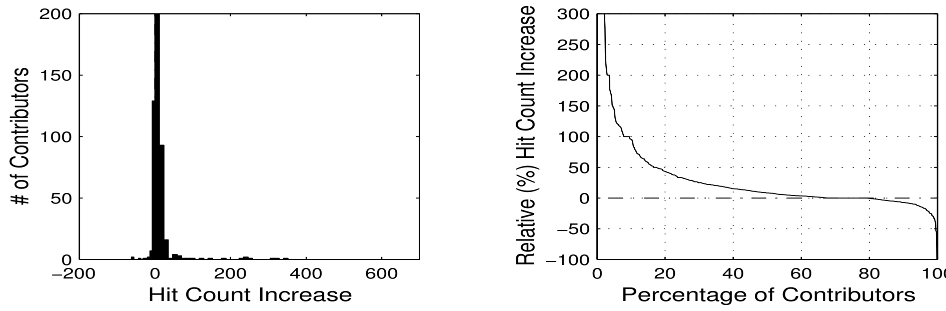


Figure 9: HPB-local Predicts More New Attacks Than GWOL-local

tacker has to target many places to be picked up by the sensors.) The LWOLs select attacker addresses that focused on the local networks. Most of these addresses had attacked far fewer contributors. HPBs's distribution is close to that of the LWOL, hence allowing the incorporation of attackers that have not saturated the Internet.

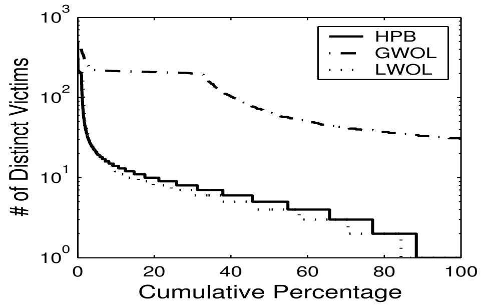


Figure 10: Cumulative Distribution of Distinct Victim Numbers

## 4.4 Performance Consistency

The results in the above experiments show that the HPB provides an increase in hit count performance across the majority of all contributors. We now ask the following question: is the HPB's performance consistent for a given contributor over time? In this experiment, we investigate this consistency question.

We use a 60-day DShield dataset. We divide it into 12 time windows, $T_0, T_1, \ldots, T_{11}$. We generate blacklists from data in time window $T_{i-1}$ and test the lists on data in $T_i$. For each contributor $v$, we compare HPB with GWOL and obtain eleven improvement values for window $T_0$ to $T_{10}$. We denote them
$IVs(v) = \{IV_0(v), IV_2(v), \ldots IV_{10}(v)\}$. We then define a consistency index (CI) for each contributor. If $IV_i(v) \geq 0$, we say that the HPB performs well for $v$ in window $i$. Otherwise, we say that the HPB performs worse. CI is the difference between the number of windows in which HPB performs well and the ones in which HPB performs poorly, i.e., $CI(v) = |\{p \in IVs(v) : p \geq 0\}| - |\{p \in IVs(v) : p < 0\}|$. If HPB consistently performs better than GWOL for a contributor, its $CI(v)$ should be close to 11. If it consistently performs worse, the $CI$ value will be close to -11. However, if the HPB performance flip-flops, its CI value will be close to zero. **Figure 11** plots the sorted CI values against the contributors. (Again, we label the x-axis by cummulative percentage.) We see that for almost 70% of the contributors, HPB's performance is extremely consistent. They all have a CI value of 11, meaning for the eleven time windows, the HPB always predicts more hits for them than GWOL. For more than 90% of the contributors, HPBs demonstrate fairly good consistency. With few contributors does the performance switch back and forth. Only 5 contributors show performance index below -3.

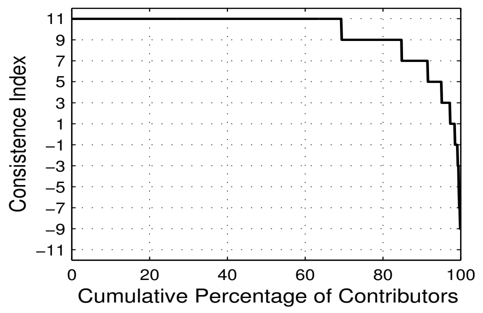


Figure 11: Cumulative Distribution of Consistency Index

The consistency investigation sheds some light on the reason why there is a small percentage of contributors for which the HPBs (sometimes) perform worse than the other list. HPB construction is based on the relevance measure. The relevance relates attack sources to contributors according to the past security logs collected by the repository. If a contributor has relatively stable correlations (stable for several days) with other contributors or it experiences stable attack patterns, the relevance measure can capture this and thus produce blacklists with more hits. Such HPBs will also be consistent in hit-count performance. On the other hand, if the correlation is not stable or the attacks exhibit few patterns, the relevance measure will be less effective and may produce blacklists with fewer hits. Such HPBs will not be consistent in performance because sometimes they may guess right and produce more hits and sometimes they may guess wrong.

This can be seen in **Figure 11**. All the consistent HPBs have CI value 11. These HPBs have both consistency and better hit-count performance. There is no HPB that shows CI value -11. HPB never performs consistently worse.

This is particularly useful because the consistency of an HPB's performance can be used to indicate whether the HPB user (the contributor) has stable correlations. If so, HPBs can be better blacklists to use. The experiment result suggests that most of the contributors have stable correlations. In practice, given a few cycles of computing HPB and GWOL for a DShield contributor, we can provide an informed recommendation as to which list that contributor should adopt over a longer term.

## 4.5 Blacklist Length

In this experiment, we vary the length of the blacklists to be 500, 1000, 5000 and 10000. We then compare the hit counts of HPBs, GWOLs, and LWOLs. Because in all the experiments, the improvements for different contributors display similar distributions, we will simply plot the medians of the hit rates of these respective blacklists. (Hit rate is the hit count divided by the blacklist length.) Our results are illustrated in **Figure 12**, and show that HPBs have the hit rate advantage for all these choices in blacklist length. The relative amount of advantage is also maintained across different lengths.

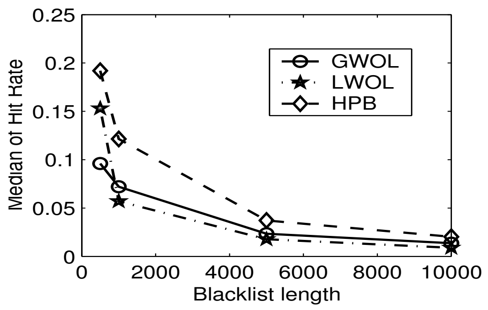


Figure 12: Hit Rates of HPB, GWOL, and LWOL with Different Lengths

Although the hit rate for the shorter lists is higher, the number of hits are larger for the longer lists. This is so for all three types of blacklists. It shows that the longer the list is, the more entries on the list are wasted (in the sense that they do not get hit). Therefore, it may not always be desirable to use very long lists.

## 4.6 Training and Prediction Window Sizes

We now investigate how far into the future the HPB can maintain its advantage over GWOL and LWOL, and how different training window sizes affect an HPB's hit count. The former helps to determine how often we need to recompute the blacklist, and the latter helps to select the right amount of history data as the input to our system. The left panel of **Figure 13** shows the median of the hit count of HPB, GWOL, and LWOL on day $1, 2, 3, \ldots, 20$ for each individual day in the prediction window. All lists are generated using data from a 5-day window prior to the prediction window. For all blacklists, the number of hits decreases along time. The HPB maintains an advantage over the entire duration of the prediction window. From this plot, we also see that the blacklists need to be refreshed frequently. In particular, there may be an almost 30% hit drop when the HPB is more than a week old.

The right panel of **Figure 13** plots hit-number medians for four HPBs. These HPBs are generated in a slightly different way from the HPBs we used so far. In previous experiments, to generate an HPB, we produce the correlation matrix from a set of attack reports. Then the sources in the same set of reports are selected into HPBs based on their relevance. In this experiment, we construct the correlation matrix using reports from training windows of size 2, 5, 7, and 10 days. Then the sources that are in the reports within the 5-day window right before the prediction (test) window are picked based on their relevance. In this formulation, we exclude sources that appear only in reports from distant history; we view their extended silence to represent a significant loss in relevance. The remainder of the test is performed in the same way as the previous experiments, i.e., the hit counts are obtained in the following 5-day prediction window. The experiment result shows that there is a slight increase in the hit counts going from a 2-day training window to a 5-day training window. The hit counts then remain roughly the same for the other training-window size. This indicates that for most of the contributors, the correlation matrix can be quite stable over time.

## 5 An Example Blacklisting Service

In mid 2007, we deployed an initial prototype implementation of the HPB system, providing a subset of the features described in this paper. This initial deployment was packaged as a free Internet blacklisting service for DShield log contributors [22,23]. HPB blacklists

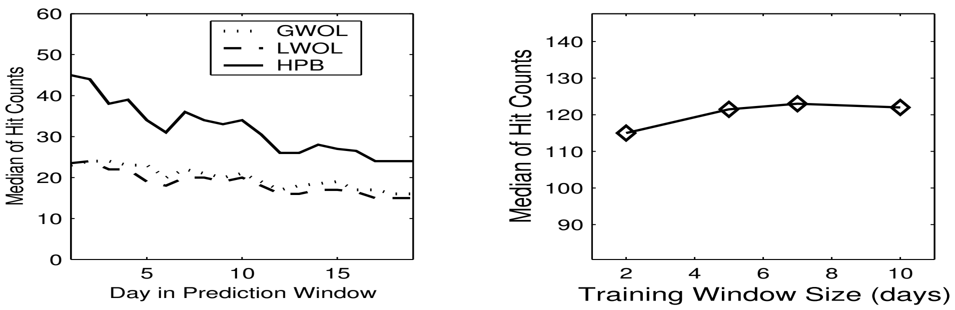


Figure 13: Effect of Training Window and Prediction Window Size on HPB's hit count

are constructed for all contributors daily, and each contributor can download her individual HPB through her DShield website account. To date, we have had a relative small pool of HPB downloaders (roughly 70 users over the most 3 months). We now describe several aspects of fielding a practical and scalable implementation of an HPB system based on our initial deployment experiences. We present an assessment of the algorithm complexity, the DShield service implementation, and discuss some open questions raised from the open release of our service.

## 5.1 Algorithm Complexity

Because HPBs are constructed from a relatively high-volume corpus of security logs, our system must be prepared to process well over 100M log entries per day to process entries over the current 5-day training window. The bottleneck of the system is the relevance ranking. Therefore, our complexity discussion focuses on the ranking algorithm. There is always an amount of complexity that is linear to the size of the alert data. That is, let $N(data)$ be the number of alerts in the data collection; we have a minimum complexity of $O(N(data))$. Our discussion will focus on other complexities incurred by the algorithm besides this linear-time requirement.

We denote by $N(s)$ and $N(v)$ the number of sources in the data collection and the number of contributors to the repository respectively. In practice, one can expect $N(v)$ to be in the order of thousands while $N(s)$ is much larger, typically in the tens of millions. We obtain $\mathbf{W}$ and $\mathbf{b}^s$ by going through the repository and doing simple accounting. The adjacency matrix $\mathbf{W}$ requires the most work to construct. To obtain this matrix, we record every overlapped attack while going through the alert data and then perform standardization. The latter steps require us to go through the whole matrix, which results in $O(N(v)^2)$ complexity.

Besides going through the data, the most time-consuming step in the relevance estimate process is the computation that solves the linear equations in **Equation 3**. At first glance, because for each source $s$, we have a linear system determined by **Equation 3**, it seems that we need to solve $N(s)$ linear systems. This can be expensive as $N(s)$ is very large. Further investigation shows that while $\mathbf{b}^s$ is different per source $s$, the $(\mathbf{I} - \mathbf{W})^{-1}$ part of the solution to **Equation 3** is the same for all $s$. Therefore, we need to compute it only once, which requires $O(N(v)^3)$ time by brute force or $O(N(v)^{2.376})$ using more sophisticated methods [5]. Because $\mathbf{b}^s$ is sparse, once we have $(\mathbf{I} - \mathbf{W})^{-1}$, the total time to obtain the ranking scores for all the sources and all the contributors is $O(N(v) \cdot N(data))$. Assuming $N(v)^2$ is much smaller than $N(data)$, the total complexity to make relevance ranking is $O(N(v) \cdot N(data))$. For a data set that contains a billion records contributed by a thousand sensors, generating a thousand rankings requires only several trillion operations (additions and multiplications). This can be easily handled by modern computers. In fact, in our experiments, with $N(data)$ in the high tens of millions and $N(v)$ on the order of one thousand, it takes less than 30 minutes to generate all contributor blacklists on an Intel Xeon 3.6 GHz machine.

## 5.2 The DShield Implementation

The pragmatics of deploying an HPB service through the DShield website are straightforward. DShield log contributors are already provided private web accounts in order to review their reports. However, to ease the automatic retrieval of HPBs, users are not required to log in via DShield's standard web account procedure. Instead, contributors wishing to access their individual HPBs can create account-specific hexadecimal tokens, and can then append this token to the HPB URL. This token has a number of advantages, particularly for developing and maintaining automated HPB retrieval scripts. That is, a user account password may be changed regularly, but the retrieval token (and script) will remain unaffected.

To provide further protection of the integrity and confidentiality of an HPB the user may also pull the HPB via

```
# DShield Customized Blocklist
# created 2007-01-19 12:13:14 UTC
# for userid 11111
# some rights reserved,  DShield Inc.,  Creative Commons Share Alike License
# License and Usage Info: http://www.dshield.org/blocklist.html
1.1.1.1      255.255.255.0        test network
2.2.2.2      255.255.255.0        another test.  This network does not exist
# End of list
```

Figure 14: A Sample Blocklist from DShield Implementation

https. A detached PGP signature can be retrieved in case https is not available or not considered a sufficient proof of authenticity.

HPBs are distributed using a simple tab-delimited format. The first column identifies the network address. The second column provides the netmask. Additional columns are used to provide more information about the respective offender, such as the name of the network and country of origin (or type of attacks seen). These additional columns are intended for human review of the HPB. Comments may be added to the blocklist. All comments start with a # mark. A sample blocklist is shown in Figure 14.

## 5.3 Gaming the System

As we have made efforts to implement, test, and advertise early versions of the HPB system, several open questions have been raised regarding the ability of adversaries to *game* the HPB system. That is, can an attacker contribute data to DShield with the intention of manipulating HPB production in ways that negatively harm HPB quality? Let us consider several questions that arise from the fact that HPBs are derived from volunteer sources, which may include dishonest contributors that are actively trying to harm or negatively manipulate HPB results.

*Can an attacker cause a consumer to incorporate an unsuspecting victim address into a third party's HPB?* Let us assume that attacker $A$ participates as one or more DShield contributors ($A$ might register multiple IDs) and knows that consumer $C$ is also a DShield contributor and an active HPB user. Furthermore, $A$ would like to cause address $B$ to be inserted into consumer $C$'s HPB. There are two potential strategies $A$ can pursue to achieve this goal. First, $A$ can spoof attacks as address $B$, directing these attacks to other contributors that are highly correlated with $A$. However, $C$'s correlated contributor set is neither readily available to $A$ (unless $A$ is a DShield insider) or necessarily stable over time. More plausibly, $A$ could artificially cause his own contributor IDs to report the same attacks as $C$. He can do this by attacking $C$ with a set of spoofed addresses, and then reporting similarly spoofed logs from his contributor IDs. Once a sufficient set of attack logs with identical spoofed attackers is reported by $C$ and $A$, $C$ could then positively influence the likelihood that address $B$ will be inserted into $A$'s HPB. While this is a possible threat, we also observe that similar attacks can be launched against GWOL and more trivially against LWOL. Furthermore, in the case of GWOL, $B$ will be inserted in **all** consumers' GWOLs, whereas A must launch this attack individually against each HPB consumer.

*Can an attacker cause his own address to be excluded from a specific third-party HPB?* Let us assume that $A$ would like to guarantee that address $B$ will not appear in $C$'s HPB. This is very difficult for $A$ to guarantee. While $A$ may cause artificial alignment between his and $C$'s logs using the alert spoofing method discussed above, $A$ cannot control what other addresses may also align with $C$. If $B$ attacks other contributors that are aligned with $C$, $B$ has the potential to enter $C$'s HPB.

*Can an attacker fully prevent or poison all HPB production?* In short, yes. Data poisoning is a fundamental threat that arises in all volunteer contributor-based data centers, and is an inherently difficult threat to overcome. However, DShield does occasionally experience, and incorporate countermeasures for issues such as *accidental* flooding and sensor misconfiguration. DDoS threats also arise and are dealt with by DShield case by case.

HPB generation could also be specifically targeted by a malicious contributor that attempts to artificially inflate the number of attacker or victim addresses, which will increase the values of $s$ or $v$, as described in our complexity analysis, Section 5.1. However, to sufficiently prohibit HPB production, the contributor would necessarily produce highly anomalous volumes of attackers (or sources) that would likely allow us to identify and (temporarily) filter this contributor.

## 6 Conclusion

In this paper, we introduced a new system to generate blacklists for contributors to a large-scale security-log sharing infrastructure. The system employs a link analysis method similar to Google's PageRank for blacklist formulation. It also integrates substantive log prefiltering and a severity metric that captures the degree to which an attacker's alert patterns match those of common malware-propagation behavior. Experimenting on a large corpus of real DShield data, we demonstrate that our blacklists have higher attacker hit rates, better *new attacker* prediction quality, and long-term performance stability.

In April of 2007, we released a highly predictive blacklist service at DShield.org. We view this service as a first experimental step toward a new direction of high-quality blacklist generation. We also believe that this service offers a new argument to help motivate the field of secure collaborative data sharing. In particular, it demonstrates that people who collaborate in blacklist formulation can share a greater understanding of attack source histories, and thereby derive more informed filtering policies. As future work, we will continue to evolve the HPB blacklisting system as our experience grows through managing the blacklist service.

## 7 Acknowledgments

This material is based upon work supported through the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316.

## References

[1] Anagnostakis, K. G., Greenwald, M. B., Ioannidis, S., Keromytis, A. D., and Li, D. A cooperative immunization system for an untrusting Internet. In *Proceedings of the 11th IEEE International Conference on Networks (ICON'03)* (October 2003).

[2] Brin, S., and Page, L. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107–117.

[3] Cai, M., Hwang, K., Kwok, Y., Song, S., and Chen, Y. Collaborative Internet worm containment. *IEEE Security and Privacy Magazine 3*, 3 (May/June 2005), 25–33.

[4] Chen, Z., and Ji, C. Optimal worm-scanning method using vulnerable-host distributions. *International Journal of Security and Networks (IJSN) Special Issue on Computer & Network Security 2*, 1 (2007).

[5] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation 9* (1990), 251–280.

[6] Humphrys, M. The Internet in the 1980s. `http://www.computing.dcu.ie/~humphrys/net.80s.html`, 2007.

[7] Incorporated, G. List of blacklists. `http://directory.google.com/Top/Computers/Internet/Abuse/Spam/Blacklist%s/`, 2007.

[8] Incorporated, G. Live-feed anti-phishing blacklist. `http://sb.google.com/safebrowsing/update?version=goog-black-url:1:1`, 2007.

[9] Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy 2004* (Oakland, CA, May 2004).

[10] Katti, S., Krishnamurthy, B., and Katabi, D. Collaborating against common enemies. In *Proceedings of the ACM SIGCOMM/USENIX Internet Measurement Conference* (October 2005).

[11] Kim, H.-A., and Karp, B. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium* (2004), pp. 271–286.

[12] Locasto, M., Parekh, J., Keromytis, A., and Stolfo, S. Towards collaborative security and P2P intrusion detection. In *Proceedings of the 2005 IEEE Workshop on Information Assurance and Security* (June 2005).

[13] M.Gori, and Pucci, A. Itemrank: A random-walk based scoring algorithm for recommender engines. In *Proceedings of the International Joint Conference on Artificial Intelligence* (January 2007).

[14] Porras, P., Briesemeister, L., Skinner, K., Levitt, K., Rowe, J., and Ting, Y. A hybrid quarantine defense. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM)* (October 2004).

[15] Ruoming, P., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. Characteristics of internet background radiation. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference* (October 2004).

[16] Thomas, R. Bogon dotted decimal list v3.9. `http://www.cymru.com/Documents/bogon-dd.hml`, October 2007.

[17] Ullrich, J. DShield global worst offender list. `https://feeds.dshield.org/block.txt`.

[18] Vixie, P., and Rand, D. Mail abuse prevention system (MAPS). `http://www.mail-abuse.com`, 1997.

[19] Wissner-Gross, A. D. Preparation of topical readings lists from the link structure of Wikipedia. In *Proceedings of the IEEE International Conference on Advanced Learning Technology* (July 2006).

[20] Yegneswaran, V., Barford, P., and Ullrich, J. Internet intrusions: global characteristics and prevalence. In *Proceedings of ACM SIGMETRICS* (June 2003).

[21] Yegneswaran, V., Porras, P., Saidi, H., Sharif, M., and Narayanan, A. The Cyber-TA compendium honeynet page. `http://www.cyber-ta.org/Honeynet`.

[22] Zhang, J., Porras, P., and Ullrich, J. The DSHIELD highly predictive blacklisting service. `http://www.dshield.org/hpbinfo.html`.

[23] Zhang, J., Porras, P., and Ullrich, J. A new service for increasing the effectiveness of network address blacklists. In *Proceedings of the 3rd Workshop of Steps to Reduce Unwanted Traffic on the Internet* (June 2007).

[24] Zhang, J., Porras, P., and Ullrich, J. Gaussian process learning for cyber-attack early warning. *to appear in Proceedings of SIAM Conference on data mining* (2008).

# Proactive Surge Protection: A Defense Mechanism for Bandwidth-Based Attacks

Jerry Chou[†], Bill Lin[†], Subhabrata Sen[‡], Oliver Spatscheck[‡]

[†]*University of California San Diego,* [‡]*AT&T Labs-Research*

***Abstract*— Large-scale bandwidth-based distributed denial-of-service (DDoS) attacks can quickly knock out substantial parts of a network before reactive defenses can respond. Even traffic flows that are not under direct attack can suffer significant *collateral damage* if these flows pass through links that are common to attack routes. Given the existence today of large botnets with more than a hundred thousand bots, the potential for a large-scale coordinated attack exists, especially given the prevalence of high-speed Internet access. This paper presents a *Proactive Surge Protection* (PSP) mechanism that aims to provide a broad first line of defense against DDoS attacks. The approach aims to minimize collateral damage by providing bandwidth isolation between traffic flows. This isolation is achieved through a combination of traffic measurements, bandwidth allocation of network resources, metering and tagging of packets at the network perimeter, and preferential dropping of packets inside the network. The proposed solution is readily deployable using existing router mechanisms and does not rely on any unauthenticated packet header information. Thus the approach is resilient to evading attack schemes that launch many seemingly legitimate TCP connections with spoofed IP addresses and port numbers. Finally, our extensive evaluation results across two large commercial backbone networks, using both distributed and targeted attack scenarios, show that up to 95.5% of the network could suffer collateral damage without protection, but our solution was able to significantly reduce the amount of collateral damage by up to 97.58% in terms of the number of packets dropped and 90.36% in terms of the number of flows with packet loss. Furthermore, we show that PSP can maintain low packet loss rates even when the intensity of attacks is increased significantly.**

## I. Introduction

A coordinated attack can potentially disable a network by flooding it with traffic. Such attacks are also known as bandwidth-based distributed denial-of-service (DDoS) attacks and are the focus of our work. Depending on the operator, the provider network may be a small-to-medium regional network or a large core network. For small-to-medium size regional networks, this type of bandwidth-based attacks has certainly disrupted service in the past. For core networks with huge capacities, one might argue that such an attack risk is remote. However, as reported in the media [6], large botnets already exist in the Internet today. These large botnets combined with the prevalence of high speed Internet access can quite easily give attackers multiple tens of Gb/s of attack capacity. Moreover, core networks are oversubscribed. For example, in the Abilene network [1], some of the core routers have an incoming capacity of larger than 30 Gb/s from the access networks, but only 20 Gb/s of outgoing capacity to the core. Although commercial ISPs do not publish their oversubscription levels, they are generally substantially higher than the ones found in the Abilene network due to commercial pressures of maximizing return on investments.

Considering these insights, one might wonder why we have not seen multiple successful bandwidth-based attacks to large core networks in the past. The answer to this question is difficult to assess. Partially, attacks might not be occurring because the organizations which control the botnets are interested in making money by distributing SPAM, committing click frauds, or extorting money from mid-sized websites. Therefore, they would have no commercial interest in disrupting the Internet as a whole. Another reason might be that network operators are closely monitoring their traffic and actively trying to intervene. Nonetheless, recent history has shown that if such an attack possibility exists, it will eventually be exploited. For example, SYN flooding attacks were described in [3] years before such attacks were used to disrupt servers in the Internet.

To defend against large bandwidth-based DDoS attacks, a number of defense mechanisms currently exist, but many are reactive in nature (i.e., they can only respond after an attack has been identified in an effort to limit the damage). However, the onset of large-scale bandwidth-based attacks can occur almost instantaneously, causing potentially a huge *surge* in traffic that can effectively knock out substantial parts of a network before reactive defense mechanisms have a chance to respond. To provide a broad first line of defense against DDoS attacks when they happen, we propose a new protection mechanism called Proactive Surge Protection (PSP). In particular, under a flooding attack, traffic loads along attack routes will exceed link capacities, causing packets to be dropped indiscriminately. Without proactive protection, even for traffic flows that are not under direct attack, substantial packet loss will occur if these flows pass through links that are common to attack routes, resulting in significant *collateral damage*. The PSP solution is based on providing *bandwidth isolation* between traffic flows so that the collateral damage to traffic flows not under direct attack is substantially reduced.

This bandwidth isolation is achieved through a combination of traffic data collection, bandwidth allocation of network capacity based on traffic measurements, metering and tagging of packets at the network perimeter into two differentiated priority classes based on capacity allocation, and preferential dropping of packets in the network when link capacities are exceeded. It is important to note that PSP has no impact on the regular operation of the network if no link is overloaded. It therefore introduces no penalty in the common case. In addition, PSP is deployable using existing router mechanisms that are already available in modern routers, which makes our approach scalable, feasible, and cost effective. Further, PSP is resilient to IP spoofing as well as changes in the underlying traffic characteristics such as the number of TCP connections. This is due to the fact that we focus on protecting traffic between different ingress-egress interface pairs in a provider network and both the ingress and egress interface of an IP datagram can be directly determined by the network operator. Therefore, the network operator does not have to rely on unauthenticated information such as a source or destination IP address to tag a packet.

The work presented in this paper substantially extends a preliminary version of our work that was initially presented at a workshop [10]. In particular, we propose a new bandwidth allocation algorithm called CDF-PSP that takes into consideration the traffic variability observed in historical traffic measurements. CDF-PSP aims to maximize in a max-min fair manner the acceptance probability (or equivalently the min-max minimization of the drop probability) of packets by using the cumulative distribution function over historical data sets as the objective function. By taking into consideration the traffic variability, we show that the effectiveness of our protection mechanism can be significantly improved. In addition, we have also substantially extended our preliminary work with much more extensive in-depth evaluation of our proposed PSP mechanism using detailed trace-driven simulations.

To test the robustness of our proposed approach, we evaluated the PSP mechanism using both *highly distributed* attack scenarios involving a high percentage of ingress and egress routers, as well as *targeted* attack scenarios in which the attacks are concentrated to a small number of egress destinations. Our extensive evaluations across two large commercial backbone networks show that up to 95.5% of the network could suffer collateral damage without protection, and our solution was able to significantly reduce the amount of collateral damage by up to 97.58% in terms of the number of packets dropped and up to 90.36% in terms of the number of flows with packet loss.

In comparison to our preliminary work, the performance of our new algorithm was able to achieve a relative reduction of up to 53.09% in terms of the number of packets dropped and up to 59.30% in terms of the number of flows with packet loss. In addition, we show that PSP can maintain low packet loss rates even when the intensity of attacks is increased significantly. Beyond evaluating extensively the impact of our protection scheme on packet drops, we also present detailed analysis on the impact of our scheme at the level of flow aggregates between individual ingress-egress interface pairs in the network.

The rest of this paper is organized as follows. Section II outlines related work. Section III presents a high-level overview of our proposed PSP approach. Section IV describes in greater details the central component of our proposed architecture that deals with bandwidth allocation policies. Section V describes our experimental setup, and Section VI presents extensive evaluation of our proposed solutions across two large backbone networks. Section VII concludes the paper.

## II. Related Work

DDoS protection has received considerable attention in the literature. The oldest approach, still heavily in use today, is typically based on coarse-grain traffic anomalies detection [21], [2]. Traceback techniques [32], [27], [28] are then used to identify the true attack source, which could be disguised by IP spoofing. After detecting the true source of the DDoS traffic the network operator can block the DDoS traffic on its ingress interfaces by configuring access control lists or by using DDoS scrubbing devices such as [4]. Although these approaches are practical, they do not allow for an instantaneous protection of the network. As implemented today, theses approaches require multiple minutes to detect and mitigate DDoS attacks, which does not match the time sensitivity of today's applications. Similarly, network management mechanisms that generally aim to find alternate routes around congested links also do not operate on a time scale that matches the time sensitivity of today's applications.

More recently, the research community has focused on enhancing the current Internet protocol and routing implementations. For example, multiple proposals have suggested to limit the best effort connectivity of the network using techniques such as capabilities models [24], [33], proof-of-work schemes [19], filtering schemes [20] or default-off communication models [7]. The main focus of these papers is the protection of customers connecting to the core network rather than protecting the core itself, which is the focus of our work. To illustrate the difference, consider a scenario in which an attacker controls a large number of zombies. These zombies could communicate with each other, granting each other capabilities or similar rights to communicate. If planned properly, this traffic is still sufficient to attack a core network. The root of the problem is that the core cannot trust either the sender or the receiver of the traffic to protect itself.

Several proactive solutions have been proposed. One solution was presented in [30]. Similar to the proposals limiting connectivity cited above, it focuses on protecting individual customers. This leads again to a trust issue in that a service provider should not trust its customers for protection. Furthermore, their solution relies heavily on the operator and customers knowing *a priori* who are the good and bad network entities, and their solution has a scalability issue in that it is not scalable to maintain detailed per-customer state for all customers within the network. Router-based defense mechanisms have also been proposed as a way to mitigate bandwidth-based attacks. They generally operate either on traffic aggregates [17] or on individual flows [22]. However, as shown in [31], these router-based mechanisms can be defeated in several ways. Moreover, deploying router-based defense mechanisms like pushback at every router can be challenging.

Our work builds on the existing body of literature on max-min fair resource allocation [8], [29], [16], [9], [25], [26], [23] to the problem of proactive DDoS defense. However, our work here is different in that we use max-min fair allocation for the purpose of differential tagging of packets with the objective of minimizing collateral damage when a DDoS attack occurs. Our work here is also different than the server-centric DDoS defense mechanism proposed in [34], which is aimed at protecting end-hosts rather than the network. In their solution, a server explicitly negotiates with selected upstream routers to throttle traffic destined to it. Max-min fairness is applied to set the throttling rates of these selected upstream routers. Like [30] discussed above, their solution also has a scalability issue in that the selected upstream routers must maintain per-customer state for the requested rate limits.

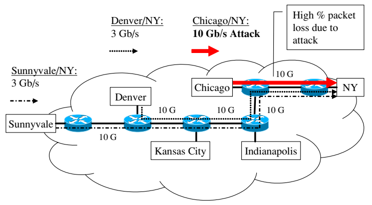
Finally, our work also builds on existing preferential dropping mechanisms that have been developed for providing Quality-of-Service (QoS) [11], [13]. However, for providing QoS, the service-level-agreements that dictate the bandwidth allocation are assumed to be either specified by customers or decided by the operator for the purpose of traffic engineering. There is also a body of work on measurement-based admission control for determining whether or not to admit new traffic into the network, e.g. [15], [18]. With both service-level-agreement-based and admission-control-based bandwidth reservation schemes, rate limits are enforced. Our work here is different in that we use preferential dropping for a different purpose to provide bandwidth isolation between traffic flows to minimize the damage that attack traffic can cause to regular traffic. Our solution is based on a combination of traffic measurements, fair bandwidth allocation, soft admission control at the network perimeter, and lazy dropping of traffic inside the network only when needed. As the mechanisms of differential tagging and preferential dropping are already available in modern routers, our solution is readily deployable.

## III. Proactive Surge Protection

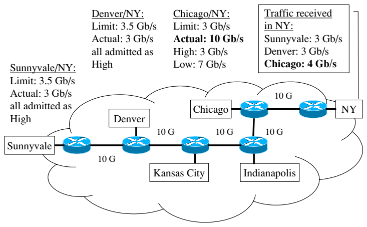
In this section, we present a high-level architectural overview of a DDoS defense solution called Proactive Surge Protection (PSP). To illustrate the basic concept, we will depict an example scenario for the Abilene network. That network consists of 11 core routers that are interconnected by OC192 (10 Gb/s) links. For the purpose of depiction, we will zoom in on a portion of the Abilene network, as shown in Figure 1(a). Consider a simple illustrative situation in which there is a sudden bandwidth-based attack along the origin-destination (OD) pair Chicago/NY, where an OD pair is defined to be the corresponding pair of ingress and egress nodes. Suppose that the magnitude of the attack traffic is 10 Gb/s. This attack traffic, when combined with the regular traffic for the OD pairs Sunnyvale/NY and Denver/NY (3 + 3 + 10 = 16 Gb/s), will significantly oversubscribe the 10 Gb/s Chicago/NY link, resulting in a high percentage of indiscriminate packet drops. Although the OD pairs Sunnyvale/NY and Denver/NY are not under *direct* attack, these flows will also suffer substantial packet loss on links which they share with the attack OD pair, resulting in significant *collateral damage*. The flows between Sunnyvale/NY and Denver/NY are said to be caught in the *crossfire* of the Chicago/NY attack.

### A. PSP Approach

The PSP approach is based on providing *bandwidth isolation* between different traffic flows so that the amount of collateral damage sustained along crossfire traffic flows is minimized. This bandwidth isolation is achieved by using a form of *soft* admission control at the perimeter of a provider network. In particular, to avoid saturation of network links, we impose *rate limits* on the amount of traffic that gets injected into the network for each OD pair. However, rather than imposing a *hard* rate limit, where packets are *blocked* from entering the network, we classify packets into two priority classes, *high* and *low*. Metering is performed at the perimeter of the network, and packets are tagged *high* if the arrival rate is below a certain threshold. But when a certain threshold is exceeded, packets will get



(a) Attack along Chicago/NY



(b) Shielded Sunnyvale/NY and Denver/NY traffic from collateral damage

Fig. 1. Attack scenario on the Abilene network.

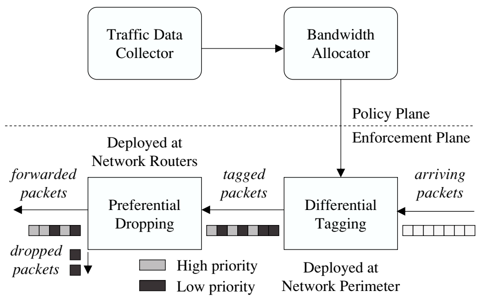


Fig. 2. Proactive Surge Protection (PSP) architecture.

tagged as *low* priority. Then, when a network link gets saturated, e.g. when an attack occurs, packets tagged with a low priority will be dropped preferentially. This ensures that our solution does not drop traffic unless a network link capacity has indeed been exceeded. Under normal network conditions, in the absence of sustained congestion, packets will get forwarded in the same manner as without our solution.

Consider again the above example, now depicted in Figure 1(b). Suppose we set the high priority rate limit for the OD pairs Sunnyvale/NY, Denver/NY, and Chicago/NY to 3.5 Gb/s, 3.5 Gb/s, and 3 Gb/s, respectively. This will ensure that the total traffic admitted as high priority on the Chicago/NY link is limited to 10 Gb/s. Operators can also set maximum rate limits to some factor below the link capacity to provide the desired headroom (e.g. set the target link load to be 90%). If the limit set for a particular OD pair is *above* the *actual* amount of traffic along that flow, then all packets for that flow will get tagged as high priority. Consider the OD pair Chicago/NY. Suppose the actual traffic under an attack is 10 Gb/s, which is above the 3 Gb/s limit. Then, only 3 Gb/s of traffic will get tagged as high priority, and 7 Gb/s will get tagged as low priority. Since the total demand on the Chicago link exceeds the 10 Gb/s link capacity, considerable packets would get dropped. However, the packets drop will come from the OD pair Chicago/NY since all packets from Sunnyvale/NY and Denver/NY would have been tagged as high priority. Therefore, the packets for the OD pairs Sunnyvale/NY and Denver/NY would be shielded from collateral damage.

Although our simple illustrative example shown in Figure 1 only involved one attack flow from one ingress point, the attack traffic in general can be highly distributed. As we shall see in Section VI, the proposed PSP method is also quite effective in such distributed attack scenarios.

## *B. PSP Architecture*

Our proposed PSP architecture is depicted in Figure 2. The architecture is divided into a policy plane and an enforcement plane. The traffic data collection and bandwidth allocation components are on the policy plane, and the differential tagging and preferential drop components are on the enforcement plane.

**Traffic Data Collector:** The role of the traffic data collection component is to collect and summarize historical traffic measurements. For example, the widely deployed Cisco sampled NetFlow mechanism can be used in conjunction with measurement methodologies such that those outlined in [14] to collect and derive traffic matrices for different times throughout a day, a week, a month, etc, between different origin-destination (OD) pairs of ingress-egress nodes. The infrastructure for this traffic data collection already exists in most service provider networks. The derived traffic matrices are used to estimate the range of expected traffic demands for different time periods.

**Bandwidth Allocator:** Given the historical traffic data collected, the role of the bandwidth allocator is to determine the *rate limits* at different time periods. For each time period $t$, the bandwidth allocator will determine a *bandwidth allocation matrix*, $B(t) = [\ b_{s,d}(t)\ ]$, where $b_{s,d}(t)$ is the rate limit for the corresponding OD pair with ingress node $s$ and egress node $d$ for a particular time of day $t$. For example, a different bandwidth allocation matrix $B(t)$ may be computed for each hour in a day using the historical traffic data collected for same hour of the day. Under normal operating conditions, network links are typically underutilized. Therefore, traffic demands from historical measurements will reflect this underutilization. Since there is likely to be *room* for admitting more traffic into the high priority class than observed in the historical measurements, we can fully allocate in some fair manner the available network resources to high priority traffic. By fully allocating the available network resources beyond the previously observed traffic, we can provide *headroom* to account for estimation inaccuracies and traffic burstiness. The bandwidth allocation matrices can be computed offline, and operators can remotely configure routers at the network perimeter with these matrices using existing router configuration mechanisms.

**Differentiated Tagging:** Given the rate limits determined by the bandwidth allocator, the role of the differential tagging component is to perform the metering and tagging of packets in accordance to the determined rate limits. This component is implemented at the perimeter of the network. In particular, packets arriving at ingress node $s$ and destined to egress node $d$ are tagged as high priority if their metered rates are below the threshold given by $b_{s,d}(t)$, using the bandwidth allocation matrix $B(t)$ for the corresponding time of day. Otherwise, they are tagged as low priority. These traffic management mechanisms for metering and tagging are commonly available in modern routers at linespeeds.

**Preferential Drops:** With packets tagged at the perimeter, low priority packets can be dropped preferentially over high priority packets at a network router whenever a sustained congestion occurs. Again, this preferential dropping mechanism [11] is commonly available in modern routers at linespeeds. By using preferential drop at interior routers rather than simply blocking packets at the perimeter when a rate limit has been reached, our solution ensures that no packet gets dropped unless a network link capacity has indeed been exceeded. Under normal network conditions, in the absence of sustained congestion, packets will get forwarded in the same manner as without our surge protection scheme.

## IV. Bandwidth Allocation Policies

Intuitively, PSP works by fully allocating the available network resources into the high priority class in some fair manner so that the high priority class rate limits for the different OD pairs are *at least* as high as the *expected* normal traffic. This way, should a DDoS attack occur that would saturate links along the attack route, *normal* traffic corresponding to *crossfire* OD pairs would be *isolated* from the attack traffic, thus minimizing collateral damage. In particular, packets for a particular crossfire OD pair would only be dropped at a congested network link if the *actual* normal traffic for that flow is *above* the bandwidth allocation threshold given to it. Therefore, bandwidth allocation plays a central role in affecting the *drop probability* of normal crossfire traffic during an attack. As such, the goal of bandwidth allocation is to allocate the available network resources with the objective of minimizing the drop probabilities for all OD pairs in some fair manner.

### A. Formulation

To achieve the objectives of minimizing drop probability and ensuring fair allocation of network resources, we formulate the bandwidth allocation problem as a utility max-min fair allocation problem [8], [9], [26], [23]. The utility max-min fair allocation problem can be stated as follows. Let $\vec{x} = (x_1, x_2, \ldots, x_N)$ be the allocation to $N$ flows, and let $(\beta_1(x_1), \beta_2(x_2), \ldots, \beta_N(x_N))$ be $N$ utility functions, with each $\beta_i(x_i)$ corresponding to the utility function for flow $i$. An allocation $\vec{x}$ is said to be *utility max-min fair* if and only if increasing one component $x_i$ must be at the expense of decreasing some other component $x_j$ such that $\beta_j(x_j) \leq \beta_i(x_i)$.

Conventionally, the literature on max-min fair allocation uses the vector notation $\vec{x}(t) = (x_1(t), x_2(t), \ldots, x_N(t))$ to represent the allocation for some time period $t$. The correspondence to our bandwidth allocation matrix $B(t) = [\ b_{s,d}(t)\ ]$ is straightforward: $b_{s_i,d_i}(t) = x_i(t)$ is the bandwidth allocation at time $t$ for flow $i$, with the corresponding OD pair of ingress and egress nodes $(s_i, d_i)$. Unless otherwise clarified, we will use the conventional vector notation $\vec{x}(t) = (x_1(t), x_2(t), \ldots, x_N(t))$ and our bandwidth allocation matrix notation interchangeably.

The utility max-min fair allocation problem has been well-studied, and as shown in [9], [26], the problem can be solved by means of a "water-filling" algorithm. We briefly outline here how the algorithm works. The basic idea is to iteratively calculate the utility max-min fair share for each flow in the network. Initially, all flows are allocated rate $x_i = 0$ and are considered free, meaning that its rate can be further increased. At each iteration, the water-filling algorithm aims to find largest increase in bandwidth allocation to free flows that will result in the maximum common utility with the available link capacities. The provided utility functions, $(\beta_1(x_1), \beta_2(x_2), \ldots, \beta_N(x_N))$, are used to determine this maximum common utility. When a link is saturated, it is removed from further consideration, and the corresponding flows that cross these saturated links are *fixed* from further increase in bandwidth allocation. The algorithm converges after at most $L$ iterations, where $L$ is the number of links in the network, since at least one new link becomes saturated in each iteration. The reader is referred to [9], [26] for detailed discussions.

In the context of PSP, the utility max-min fair algorithm is used to implement different bandwidth allocation policies. In particular, we describe in this section two bandwidth allocation policies, one called Mean-PSP, and the other called CDF-PSP. Both are based on traffic data collected from historical traffic measurements. The first policy, Mean-PSP, simply uses the average historical traffic demands observed as *weights* in the corresponding utility functions. Mean-PSP is based

TABLE I
TRAFFIC DEMANDS AND THE CORRESPONDING BANDWIDTH ALLOCATIONS FOR MEAN-PSP AND CDF-PSP.

| Flows | Historical traffic measurements | | | | | | BW allocation | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Measured demands (sorted) | | | | | Mean | Mean-PSP | | CDF-PSP | |
| | | | | | | | 1st | 2nd | 1st | 2nd |
| (A,D) | 1 | 1 | 2 | 2 | 4 | 2 | 2 | 2 | 2 | 2 |
| (B,D) | 1 | 1 | 1 | 3 | 4 | 2 | 2 | 2 | 3 | 3 |
| (C,D) | 4 | 5 | 5 | 5 | 11 | 6 | 6 | 6 | 5 | 5 |
| (A,C) | 4 | 5 | 5 | 5 | 11 | 6 | 6 | 8 | 5 | 8 |
| (B,C) | 5 | 5 | 6 | 6 | 8 | 6 | 6 | 8 | 6 | 7 |

on the simple intuition that flows with higher average traffic demands should receive proportionally higher bandwidth allocation. This policy was first presented in our preliminary work [10]. However, this policy does not directly consider the traffic variance observed in the traffic measurements.

To directly account for traffic variance, we propose a second policy, CDF-PSP, that explicitly aims to minimize drop probabilities by using the *Cumulative Distribution Functions* (CDFs) [8] derived from the empirical distribution of traffic demands observed in the traffic measurements. These CDFs can be used to capture the probability that the actual traffic will not exceed a particular bandwidth allocation. When these CDFs are used as utility functions, maximizing the utility corresponds directly to the minimization of drop probabilities. Each of these two policies is further illustrated next.

## B. Mean-PSP: Mean-based Max-min Fairness

Our first allocation policy, Mean-PSP, simply uses the mean traffic demand as the utility function. In particular, the utility function for flow $i$ is a simple linear function $\beta_i(x) = \frac{x}{\mu_i}$, where $\mu_i$ is the mean traffic demand of flow $i$, which simplifies to an easier weighted max-min fair allocation problem.

To illustrate how Mean-PSP works, consider the small example shown in Figure 3. It depicts a simple network topology with 4 nodes that are interconnected by 10 Gb/s links. Consider the corresponding traffic measurements shown in Table I. For simplicity of illustration, each flow is described by just 5 data points, and the corresponding mean traffic demands are also indicated in Table I. Consider the first iteration of the Mean-PSP water-filling procedure shown in Figure 4(a). The maximum common utility that can be achieved by all *free* flows is $\beta(x) = 1$, which corresponds to allocating 2 Gb/s each to the OD pairs $(A, D)$ and $(B, D)$ and 6 Gb/s each to the OD pairs $(C, D)$, $(A, C)$, and $(B, C)$. For example, $\beta_{A,D}(x) = \frac{x}{\mu} = 1$ corresponds to allocating $x = 2$ Gb/s since $\mu$ for $(A, D)$ is 2. Since all three flows, $(A, D)$, $(B, D)$, and $(C, D)$, share a common link $CD$, the sum of their first iteration allocation, 2 + 2 + 6 = 10 Gb/s, would already saturate link $CD$. This saturated link is removed from consideration in subsequent iterations, and the flows $(A, D)$, $(B, D)$, and $(C, D)$ are fixed at the allocation of 2 Gb/s, 2 Gb/s, and 6 Gb/s, respectively.

On the other hand, link $AC$ is only shared by flows $(A, C)$ and $(A, D)$, which has an aggregate allocation of 2 + 6 = 8 Gb/s on link $AC$ after the first iteration. This leaves $10 - 8 = 2$ Gb/s of *residual* capacity for the next iteration. Similarly, link $BC$ is only shared by flows $(B, C)$ and $(B, D)$, which also has an aggregate allocation of 2 + 6 = 8 Gb/s on link $BC$ after the first iteration, with 2 Gb/s of residual capacity. After the first iteration, flows $(A, C)$ and $(B, C)$ remain free.

In the second iteration, as in shown Figure 4(b), the maximum common utility is achieved by allocating the remaining 2 Gb/s on link $AC$ to flow $(A, C)$ and the remaining 2 Gb/s on link $BC$ to flow $(B, C)$, resulting in each flow having 8 Gb/s allocated to it in total. The final Mean-PSP bandwidth allocation is shown in Table I.

## C. CDF-PSP: CDF-based Max-min Fairness

Our second allocation policy, CDF-PSP, aims to explicitly capture the *traffic variance* observed in historical traffic measurements by using a Cumulative Distribution Function (CDF) model as the utility function. In particular, the use of CDFs [8] captures the *acceptance probability* of a particular bandwidth allocation as follows. Let $X_i(t)$ be a random variable that represents the *actual* normal traffic for flow $i$ at time $t$, and let $x_i(t)$ be the bandwidth allocation. Then the CDF of $X_i(t)$ is denoted as

$$Pr[X_i(t) \leq x_i(t)] = \Phi_{i,t}(x_i(t)),$$

and the drop probability is simply the complementary function

$$Pr[X_i(t) > x_i(t)] = 1 - \Phi_{i,t}(x_i(t)).$$

Therefore, when CDFs are used to maximize the acceptance probabilities for all flows in a max-min fair manner, it is equivalent to minimizing the drop probabilities for all flows in a min-max fair manner.

In general, the expected traffic can be modeled using different probability density functions with the corresponding CDFs. One probability density function is to use the empirical distribution that directly corresponds to the historical traffic measurements taken. In particular, let $(r_{i,1}(t), r_{i,2}(t), \ldots, r_{i,M}(t))$ be $M$ measurements taken for flow $i$ at a particular time of day $t$ over some historical data set. Then the empirical CDF is simply defined as

$$\Phi_{i,t}(x_i(t)) = \frac{\text{\# measurements } \leq x_i(t)}{M}$$

$$= \frac{1}{M} = \sum_{k=1}^{M} I(r_{i,k}(t) \leq x_i(t)),$$

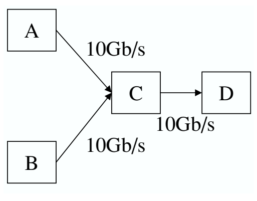


Fig. 3. Network.

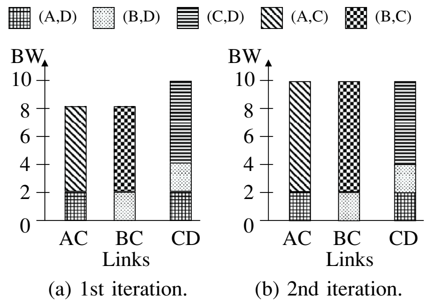


Fig. 4. Mean-PSP water-filling illustrated.

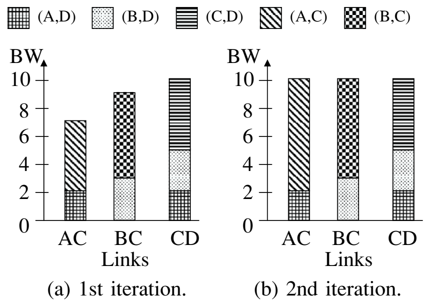


Fig. 5. CDF-PSP water-filling illustrated.

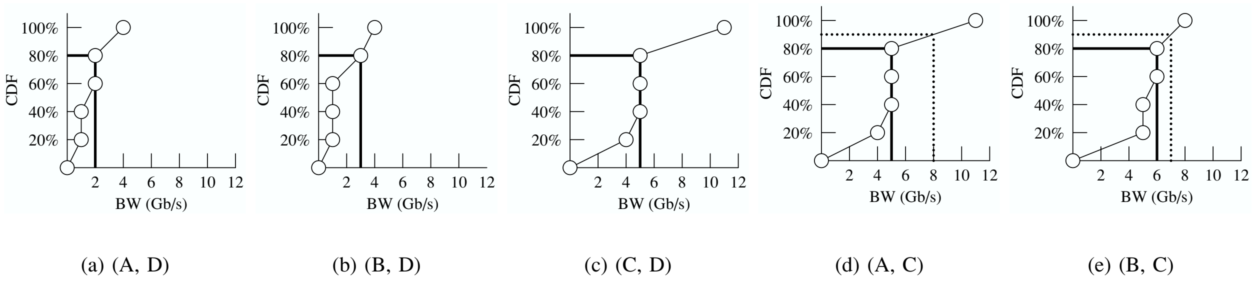


Fig. 6. Empirical CDFs for flows (A, D), (B, D), (C, D), (A, C), (B, C).

where $I(r_{i,k}(t) \leq x_i(t))$ is the indicator that the measurement $r_{i,k}(t)$ is less than or equal to $x_i(t)$. For the example shown in Table I, the corresponding empirical CDFs are shown in Figure 6. For example in Figure 6(a) for OD pair $(A, D)$, a bandwidth allocation of 2 Gb/s would correspond to an acceptance probability of 80% (with the corresponding drop probability of 20%).

To illustrate how CDF-PSP works, consider again the example shown in Figure 3 and Table I. Consider the first iteration of the CDF-PSP water-filling procedure shown in Figure 5(a). To simplify notation, we will simply use for example $\beta_{A,D}(x) = \Phi_{A,D}(x)$ to indicate the utility function for flow $(A, D)$ for some time period $t$, and we will use analogous notations for the other flows.

In the first iteration, the maximum common utility that can be achieved by all free flows is an acceptance probability of $\beta(x) = 80\%$, which corresponds to allocating 2 Gb/s to $(A, D)$, 3 Gb/s to $(B, D)$, 5 Gb/s each to $(C, D)$ and $(A, C)$, and 6 Gb/s to $(B, C)$. This first iteration allocation is shown in bold black lines in Figure 6. With this allocation in the first iteration, link $CD$ is again saturated since the sum of the first iteration allocation to flows $(A, D)$, $(B, D)$, and $(C, D)$ is 2 + 3 + 5 = 10 Gb/s, which would already reach the link capacity of $CD$. Therefore, the saturated link $CD$ is removed from consideration in subsequent iterations, and the flows $(A, D)$, $(B, D)$, and $(C, D)$ are fixed at the allocation of 2 Gb/s, 3 Gb/s, and 5 Gb/s, respectively.

For link $AC$, which is shared by flows $(A, C)$ and $(A, D)$, the first iteration allocation is 2 + 5 = 7 Gb/s, leaving 10−7 = 3 Gb/s of residual capacity. Similarly, for link $BC$, which is shared by flows $(B, C)$ and $(B, D)$, the first iteration allocation is 3 + 6 = 9 Gb/s, leaving 10 − 9 = 1 Gb/s of residual capacity.

In the second iteration, as in shown Figure 5(b), the maximum common utility 90% is achieved for the remaining free flows $(A, C)$ and $(B, C)$ by allocating the remaining 3 Gb/s on link $AC$ to flow $(A, C)$ and the remaining 1 Gb/s on link $BC$ to flow $(B, C)$, resulting in a total of 8 Gb/s allocated to $(A, C)$ and 7 Gb/s allocated to $(B, C)$. This second iteration allocation is shown in dotted lines in Figure 6. The final CDF-PSP bandwidth allocation is shown in Table I.

Comparing the results for CDF-PSP and Mean-PSP shown in Figure 6 and Table I, we see that CDF-PSP was able to achieve a higher worst-case acceptance probability for all flows than Mean-PSP. In particular, the CDF-PSP results shown in Figure 6 and Table I show that CDF-PSP was able to achieve a minimum acceptance probability of 80% for all flows whereas Mean-PSP was only able to achieve a lower worst-case acceptance probability of 70%. For example, for flow $(B, D)$, the bandwidth allocation of 3 Gb/s determined by CDF-PSP corresponds to an 80% acceptance rate whereas the 2 Gb/s determined by Mean-PSP only corresponds to a 70% acceptance rate. The better worst-case result is because CDF-PSP specifically targets the max-min optimization of the *acceptance probability* by using the cumulative distribution function as the objective.

## V. Experimental Setup

We employed ns-2 based simulations to evaluate our PSP methods on two large real networks.

**US:** This is the backbone of a large service provider in the US, and consists of around 700 routers and thousands of links ranging from T1 to OC768 speeds.

**EU:** This is the backbone of a large service provider in Europe. It has a similar network structure as the US backbone, but it is larger with about 150 more routers and 500 more links.

While the results for the individual networks cannot be directly compared to each other because of differences in their network characteristics and traffic behavior, multiple network environments allow us to explore and understand the performance of our PSP methods for a range of diverse scenarios.

### A. Normal Traffic Demand

For each network, using the methods outlined in [14], we build ingress router to egress router traffic matrices from several weeks worth of sampled Netflow data that record the traffic for that network : US (07/01/07−09/03/07) and EU (11/18/06−12/18/06 & 07/01/07−09/03/07). Specifically, the Netflow data contains sampled Netflow records covering the entire network. The sampling is performed on the routers with 1:500 packet sampling rate. The volume of sampled records are then subsequently reduced using a smart sampling technique [12]. The total size of smart sampled data records was 3,600 GB and 1,500 GB for US and EU, respectively. Finally, we annotate each record with its customer egress interface (if it was not collected on the egress router) based on route information.

For each time interval $\tau$, the corresponding OD flows are represented by a $N \times N$ traffic matrix where $N$ is the number of access routers providing ingress or egress to the backbone, and each entry contains the average demand between the corresponding routers within that interval. The above traffic data are used both for creating the normal traffic demand for the simulator as well as for computing the corresponding bandwidth allocation matrices for the candidate PSP techniques. One desirable characteristic from a network management, operations and system overhead perspective is to avoid too many unnecessary fine time scale changes. Therefore, one goal of our study was to evaluate the effectiveness of using a single representative bandwidth allocation matrix for an extended period of time. An implicit hypothesis is that the bandwidth allocation matrix does not need to be computed and updated on a fine timescale. To this end, in the simulations, we use a finer timescale traffic matrix with $\tau = 1$ min for determining the normal traffic demand, and a coarser timescale 1 hour interval for computing the bandwidth allocation matrix from historical data sets.

### B. DDoS Attack Traffic

To test the robustness of our PSP approach, we used two different types of attack scenarios for evaluation – a *distributed* attack scenario for the US backbone and a *targeted* attack scenario for the EU backbone. As we shall see in Section VI, PSP is very effective in both types of attacks. In particular, we used the following attack data.

**US DDoS:** For the US backbone, the attack matrix that we used for evaluation is based on large DDoS alarms that were actually generated by a commercial DDoS detection system deployed at key locations in the network. In particular, among the actual large DDoS alarms there were generated during the period of 6/1/05 to 7/1/06, we selected the largest one involving the most number of attack flows as the attack matrix. This was a *highly distributed* attack involving 40% (nearly half) of the ingress routers as attack sources and 25% of the egress routers as attack destinations. The number of attack flows observed at a single ingress router were up to 150 flows, with an average of about 24 attack flows sourced at each ingress router. The attacks were distributed over a large number of egress routers. Although the actual attacks were large enough to trigger the DDoS alarms, they did not actually cause overloading on any backbone link. Therefore, we scaled up each attack flow to an average of 1% of the ingress router link access capacity. Since there were many flows, this was already sufficient to cause overloading on the network.

**EU DDoS:** For the Europe backbone, we had no commercial DDoS detection logs available. Therefore, we created our own synthetic DDoS attack data. To evaluate PSP under different attack scenarios, we created a *targeted* attack scenario in which all attack flows are targeted to only a small number of egress routers. In particular, to mimic the US DDoS attack data, we randomly selected 40% of ingress routers to be attack sources. However, to create a targeted attack scenario, we purposely selected at random only 2% of the egress routers as attack destinations. With only 2% of the egress routers involved as attack destinations, we concentrated the attacks from each ingress router to just 1-3 destinations with demand set at 10% of the ingress router link access capacity.

### C. ns-2 Simulation Details

Our experiments are implemented using ns-2 simulations. This involved implementing the 2-class bandwidth allocation, and simulating both the normal and DDoS traffic flows.

**Bandwidth Allocation and Enforcement:** The metering and class differentiation of packets are implemented at the perimeter of each network using the differentiated service module in ns-2, which allows users to set rate limits for each individual OD pair. Our simulation updates the rate limits hourly by pre-computing the bandwidth allocation matrix based on the historical traffic matrices that were collected several weeks prior to the attack date: US (07/01/07−09/02/07) and EU (11/18/06−12/17/06 & 07/01/07−09/02/07).

The differentiated service module marks incoming packets into different priorities based on the configured rate limits set by our bandwidth allocation matrix and the estimated incoming traffic rate of the OD pair. Specifically, we implemented differentiated service using TSW2CM (Time Sliding Window with 2 Color Marking), an ns-2 provided policer. As its name implies, the TSW2CM policer uses a sliding time window to estimate the traffic rate.

If the estimated traffic exceeds the given threshold, the incoming packet is marked into the low priority class; otherwise, it is marked into the high priority class. We then use existing preferential dropping mechanisms to ensure that lower priority packets are preferentially dropped over higher priority packets when memory buffers get full. In particular, WRED/RIO[1] is one such preferential dropping mechanism that is widely deployed in existing commercial routers [11], [5]. We used this WRED/RIO mechanism in our ns-2 simulations.

**Traffic Simulation:** For simulation data (testing phase), we purposely used a different data set than the traffic matrices used for bandwidth allocation (learning phase). In particular, for each network, we selected a week-day outside of the days used for bandwidth allocation, and we considered 48 1-minute time intervals (one every 30-minutes) across the entire 24 hours of this selected day. The exact date that we selected to simulate normal traffic is 09/03/07 for both the US and EU networks. Recall that for a given time interval $\tau$, we compute normal and DDoS traffic matrices that give average traffic rates across that interval. These matrices are used to generate the traffic flows for that time interval. Both DDoS and network traffic are simulated as constant bandwidth UDP streams with fixed packet sizes of 1 kB.

## VI. Experimental Results

We begin our evaluations in Section VI-A by quantifying the potential extent and severity of the problem that we are trying to address – the amount of collateral damage in each network in the absence of any protection mechanism. We then develop an understanding of the damage mitigation capabilities and properties of our PSP mechanism, first at the network level in Section VI-B and then at the individual OD-pair level in Section VI-C. Section VI-D explores the effectiveness of the proposed schemes under scaled attacks, and Section VI-E summarizes all the results.

We shall use the term No-PSP to refer to the baseline scenario with no surge protection. We use the terms Mean-PSP and CDF-PSP to refer to the PSP schemes that use proportional and empirical CDF-based water-filling bandwidth allocation algorithms respectively. Recall that an OD pair is considered as (i) an ***attacked OD pair*** if there is attack traffic along that pair, (ii) a ***crossfire OD pair*** if it shares at least one link with an OD pair containing attack traffic, and (iii) a ***non-crossfire OD pair*** if it is neither an *attacked* nor a *crossfire* OD pair.

### A. Potential for Collateral Damage

We first explore the extent to which OD pairs and their offered traffic demands are placed in potential harm's way because they share network path segments with a given set of attack flows. In Figure 7, we report the relative proportion of OD pairs in the categories of *attacked*, *crossfire*, and *non-crossfire* OD pairs for both the US and EU backbones.

As described in Section V-B, 40% of the ingress routers and 25% of the egress routers were involved in the DDoS attack on the US backbone. In general, for a network with $N$ ingress/egress routers, there are $N^2$ possible OD pairs (the ratio of routers to OD pairs is 1-to-$N$). For the US backbone, with about 700 routers, there are nearly half a million OD pairs. Although 40% of the ingress routers and 25% of the egress routers were involved in the attack, the number of attack destinations from each ingress router was on average about 24 egress routers, resulting in just 1.2% of the OD pairs under direct attack. In general, because the number of OD pairs grows quadratically with $N$ (i.e. $N^2$), even in a highly distributed attack scenario where the attack flows come from all $N$ routers, the number of OD pairs under direct attack may still only correspond to a small percentage of OD pairs. For the EU backbone, there are about 850 routers and about three quarters of million OD pairs. For the targeted attack scenario described in Section V-B, 40% of the ingress routers were also involved in the DDoS attack, but the attacks were concentrated to just 2% of the egress routers. Again, even though 40% of the ingress routers were involved, only 0.1% of the OD pairs, among $N^2$ OD pairs, were under direct attack.

In general, the percentage of OD pairs that are in the crossfire of attack flows depends on where the attacks occurred and how traffic is routed over a particular network. For the US backbone, we observe that the percentage of crossfire OD pairs is very large (95.5%),

[1]RIO is WRED with two priority classes.

TABLE II

COLLATERAL DAMAGE IN THE ABSENCE OF PSP WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

| | Impacted OD Pairs(%) | Impacted Demand(%) | Mean packet loss rate of impacted OD pairs(%) |
|---|---|---|---|
| US | 41.37 [39.64, 42.72] | 37.79 [35.16, 39.37] | 49.15 [47.62, 50.43] |
| EU | 43.18 [38.48, 47.81] | 45.33 [38.90, 52.05] | 68.11 [65.51, 70.46] |

causing substantial collateral damage even though the attacks were directed over only 1.2% the OD pairs. This is somewhat expected given the distributed nature of the attack where a high percentage of both ingress and egress routers were involved in the attack. For the EU backbones, the observed percentage of crossfire OD pairs is also very large (83.5%). This is somewhat surprisingly because the attacks were targeted to only a small number of egress routers. This large footprint can be attributed to the fact that even a relatively small number of attack flows can go over common links that were shared by a vast majority of other OD pairs.

We next depict the relative proportions of the overall normal traffic demand corresponding to each type of OD pairs. While the classification of the OD pairs into the 3 categories is fixed for a given network and attack matrix, the relative traffic demand for the different classes is time-varying, depending on the actual normal traffic demand in a given time interval. Figure 8 presents a breakdown of the total normal traffic demands for the 3 classes across the 48 time intervals that we explored. Note that for both the networks, crossfire OD pairs account for a significant proportion of the total traffic demand. Figures 7 and 8 together suggest that an attack directed even over a relatively small number of ingress-egress interface combinations, could be routed around the network in a manner that can impact a significant proportion of OD pairs and overall network traffic.

The results above provide us an indication of the potential "worst-case" impact footprint that an attack can unleash, if its strength is sufficiently scaled up. This is because a crossfire OD pair will suffer collateral packet losses only if some link(s) on its path get congested. While the above results do not provide any measure of actual damage impact, they do nevertheless point to the existence of a real potential for widespread collateral damage, and underline the importance and urgency of developing techniques to mitigate and minimize the extent of such damage.

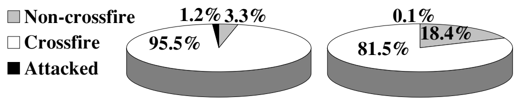
We next consider the actual collateral damage induced by the specified attacks in the absence of any protection scheme. We define a crossfire OD pair to be *impacted* in a given time interval, if it suffered some packet loss in that interval. Table II presents (i) the total number of, and (ii) traffic demand for the impacted OD pairs as a percentage of the corresponding values for all crossfire OD pairs, and (iii) the mean packet loss rate across the impacted OD pairs. To account for time variability, we present the average value (with the $10^{th}$ and $90^{th}$ percentile indicated in the brackets) for the three metrics across the 48 attacked time intervals. Overall, the tables show that not only can the attacks impact a significant proportion of the crossfire OD pairs and network traffic, but that they can cause severe packet drops in many of them. For example, in the EU network, in $90\%$ of the time intervals, (i) at least $39.64\%$ of the cross-fire OD pairs were impacted, and (ii) the average packet loss rate across the impacted OD pairs was $47.62\%$ or more. To put these numbers in proper context, note that TCP, which accounts for the vast majority of traffic today, is known to have severe performance problems once the loss rate exceeds a few single-digit percentage points.

## *B. Network-wide PSP Performance Evaluation*

We start the evaluation of PSP by focusing on network-wide aggregate performance for crossfire OD pairs and note the consistent substantially lower loss rates under either Mean-PSP or CDF-PSP across the entire day.

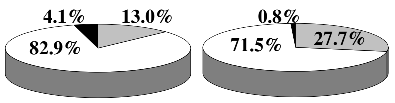
*1) Total Packet Loss Rate:*

For each attack time interval, we compute the ***total packet loss rate*** which is the total number of packets lost as a percentage of the total offered load from all crossfire OD pairs. Table III summarizes the mean, $10^{th}$ and $90^{th}$ percentile of the total packet loss rates across 48 attack time intervals. The mean loss rates under No-PSP in US and EU networks are 17.93% and 30.48%, respectively. The loss rate is relatively stable across time as indicated by the tight interval between the $10^{th}$ and $90^{th}$ percentile numbers. In contrast, the mean loss rate is much smaller, less than 3%, for either PSP scheme. Figure 9 shows the loss rate across time, for the 2 PSP schemes, expressed as a percentage of the corresponding loss rates under No-PSP. Note that even though the attack remains the same over all 48 attack time intervals, the normal traffic demand matrix is time-varying, and hence the observed variability in the time series. In particular, we observe comparatively smaller improvements during the the network traffic peak times, such as 12PM (GMT) in the EU backbone and 6PM (GMT) in the US backbone. This behavior is because the amount of traffic that could be admitted as high priority is bounded by the network's carrying capacity. During high demand time intervals, on one hand, links will be more loaded increasing the likelihood of congestion and overload. On the other hand, more packets will get classified as low priority, increasing the population size that can be dropped under congestion and overload. Table IV



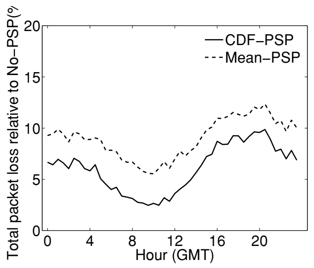
(a) US. (b) Europe.

Fig. 7. The percentage of the number of the three OD pair types classified under an attack traffic.

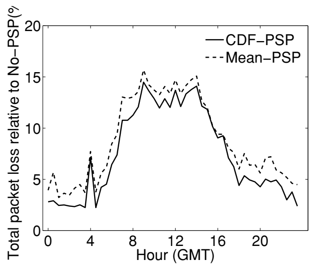


(a) US. (b) Europe.

Fig. 8. The proportion of normal traffic demand corresponding to the three types of OD pairs.



(a) US



(b) EU

Fig. 9. The crossfire OD pair total packet loss rate ratio over No-PSP across 24 hours.(48 attack time intervals, 30 minutes apart).

TABLE III
THE TIME-AVERAGED CROSSFIRE OD-PAIR TOTAL PACKET LOSS RATE WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

| | No-PSP | Mean-PSP | CDF-PSP |
|---|---|---|---|
| US | 17.93 [16.40, 18.79] | 1.63 [1.02, 2.14] | 1.11 [0.47, 1.71] |
| EU | 30.48 [27.22, 32.86] | 2.73 [1.21, 4.54] | 2.32 [0.79, 4.22] |

TABLE IV
THE TIME-AVERAGED TOTAL PACKET LOSS REDUCTION RELATIVE TO NO-PSP OR MEAN-PSP WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

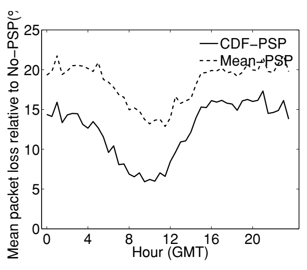
| | Reduction ratio from No-PSP to Mean-PSP | Reduction ratio from No-PSP to CDF-PSP | Reduction ratio from Mean-PSP to CDF-PSP |
|---|---|---|---|
| US | 91.00 [88.56, 93.89] | 93.90 [90.77, 97.21] | 34.75 [20.06, 53.09] |
| EU | 91.17 [85.79, 96.17] | 92.51 [86.46, 97.58] | 19.90 [4.01, 41.58] |

summarizes the performance improvements for the PSP schemes in terms of relative loss rate reduction to No-PSP or Mean-PSP across the different time intervals. For each network, on average, either PSP scheme reduces the loss rate in a time interval by more than $90\%$ from the corresponding No-PSP value. In addition CDF-PSP has consistently better performance than Mean-PSP with loss rates that are on average $34.75\%$ and $19.90\%$ lower for the US and EU networks, respectively.

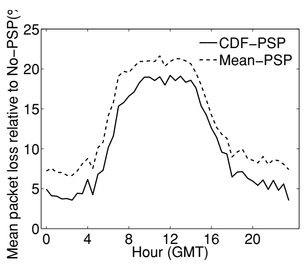
*2) Mean OD Packet Loss Rate:*

Our second metric is the *mean OD packet loss rate* which measures the average packet loss rate across all crossfire OD pairs with non-zero traffic demand. For each of the 48 attack time intervals, for each crossfire OD pair that had traffic demand in that interval, we compute its ***packet loss rate***, ie., the number of packets dropped as a percentage of its total offered load. The mean OD packet loss rate is obtained by averaging across these per-OD pair loss rates for that interval. Table V presents the average, $10^{th}$ and $90^{th}$ percentile values for that metric across the 48 time intervals for the different PSP scenarios. Figure 10 shows the time series of the metric for Mean-PSP and CDF-PSP, expressed as a percentage of the corresponding value for No-PSP. The table and the figure clearly show that, across time, No-PSP had consistently much higher mean OD packet loss rate than Mean-PSP and CDF-PSP, while CDF-PSP has the best performance. The percentage improvements are summarized in Table VI, which show that going from No-PSP to CDF-PSP results in a reduction in the mean OD packet loss rate by 87.50% and 89.93% for the US and EU networks, respectively. Moving from Mean-PSP to CDF-PSP reduces this loss rate metric by 33.20% and 25.46% respectively in the two networks.

*3) Number of impacted crossfire OD pairs:* We next determine the number of impacted OD pairs, ie., the crossfire OD pairs that suffer some packet loss at each time interval. It is desirable to minimize this number, since many important network applications including real-time gaming and VOIP are very sensitive to and experience substantial performance degradations even under relatively low packet loss rates. For each of the 48 attack time intervals, we determine the number of impacted crossfire OD pairs as a percentage of the

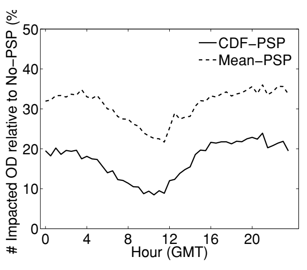


(a) US

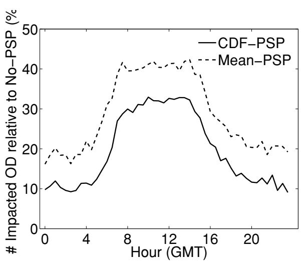


(b) EU

Fig. 10. The mean OD packet loss rate ratio over No-PSP across 24 hours.(48 attack time intervals, 30 minutes apart).



(a) US



(b) EU

Fig. 11. The ratio of number of crossfire OD-pairs with packet loss over No-PSP across 24 hours.(48 attack time intervals, 30 minutes apart).

TABLE V

THE TIME-AVERAGED CROSSFIRE OD-PAIR MEAN PACKET LOSS RATE. THE $10^{th}$ AND $90^{th}$ PERCENTILE NUMBER ARE INDICATED IN THE BRACKETS.

| | No-PSP | Mean-PSP | CDF-PSP |
|---|---|---|---|
| US | 20.33 [19.25, 21.07] | 3.75 [2.69, 4.31] | 2.56 [1.33, 3.39] |
| EU | 29.34 [26.62, 32.16] | 4.04 [2.02, 6.71] | 3.23 [1.09, 5.98] |

TABLE VI

THE TIME-AVERAGED CROSSFIRE OD-PAIR MEAN PACKET LOSS RATE REDUCTION RELATIVE TO NO-PSP AND MEAN-PSP WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

| | Reduction ratio from No-PSP to Mean-PSP | Reduction ratio from No-PSP to CDF-PSP | Reduction ratio from Mean-PSP to CDF-PSP |
|---|---|---|---|
| US | 81.65 [79.27, 86.19] | 87.50 [83.88, 93.33] | 33.20 [19.65, 52.84] |
| EU | 86.63 [79.01, 92.77] | 89.39 [81.15, 95.92] | 25.46 [9.83, 44.94] |

TABLE VII

THE TIME-AVERAGED NUMBER OF IMPACTED OD-PAIRS WITH PACKET LOSS WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

| | No-PSP | Mean-PSP | CDF-PSP |
|---|---|---|---|
| US | 41.37 [39.06, 42.73] | 12.85 [9.58, 14.58] | 7.16 [3.94, 9.24] |
| EU | 43.18 [38.43, 47.94] | 12.81 [7.28, 19.70] | 8.79 [3.84, 15.46] |

TABLE VIII

THE TIME-AVERAGED REDUCTION OF NUMBER OF IMPACTED OD-PAIRS WITH PACKET LOSS RELATIVE TO NO-PSP AND MEAN-PSP WITH THE $10^{th}$ AND $90^{th}$ PERCENTILE INDICATED IN THE BRACKETS.

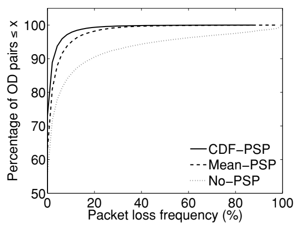
| | Reduction ratio from No-PSP to CDF-PSP | Reduction ratio from No-PSP to CDF-PSP | Reduction ratio from Mean-PSP to CDF-PSP |
|---|---|---|---|
| US | 69.05 [65.20, 75.64] | 82.82 [78.11, 90.22] | 45.47 [35.12, 59.30] |
| EU | 71.18 [58.62, 81.49] | 80.42 [67.66, 90.36] | 34.94 [21.72, 47.60] |

total number of crossfire OD pairs with non-zero traffic demand in that time interval. We summarize the mean and the $10^{th}$ and $90^{th}$ percentiles from the distribution of the resulting values across the 48 time intervals in Table VII for No-PSP and the two PSP schemes. The mean proportion of impacted OD pairs drops from a high of $41.37\%$ under No-PSP to $12.85\%$ for No-PSP to $7.16\%$ for CDP-PSP. We present the time series of the proportion of impacted OD pairs for the two PSP schemes (normalized by the corresponding value for No-PSP) across the 48 time intervals in Figure 11, and summarize the savings from the 2 PSP schemes in Table VIII. Across all the time intervals, we note that a high percentage of crossfire OD pairs had packet losses under-No-PSP, and that both PSP schemes dramatically reduce this proportion, with CDF-PSP consistently having the lowest proportion of impacted OD pairs. Considering the Table VIII , the proportion of impacted OD pairs in the US network is reduced, on average, by over $69\%$ going from No-PSP to Mean-PSP. From Mean-PSP to CDF-PSP, the proportion drops, on average, by a further substantial $45.47\%$.

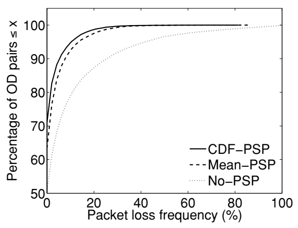
## C. OD pair-level Performance

In Section VI-B, we explored the performance of the PSP techniques from the overall network perspective. We focus the analysis below on the performance of individual crossfire OD pairs across time.

*1) Loss Frequency:* For each crossfire OD pair, we define its ***loss frequency*** to be the percentage of of the 48 attack time intervals in which it incurred some packet loss. Note that this metric only captures how often across the different times of day, a crossfire OD pair experiences loss events, and is not meant to capture the actual magnitude of individual loss events which we shall study later. Figure 12 plots the cumulative

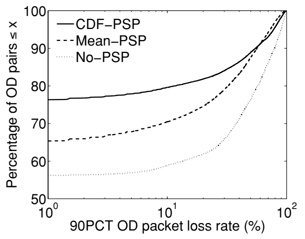


(a) US

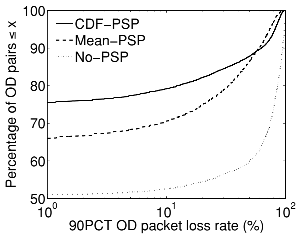


(b) EU

Fig. 12. CDF of the loss frequency for all crossfire OD pairs.



(a) US



(b) EU

Fig. 13. CDF of the 90 percentile packet loss rate for all crossfire OD pairs.

distribution function (CDF) of the loss frequencies across all the crossfire OD pairs which had some traffic in any of the 48 intervals. In the figure, a given point $(x, y)$ indicates that $y$ percent of crossfire OD-pairs had packet loss in at most $x$ percent of the attack time intervals. Therefore, corresponding to the same $x$ value, the larger the $y$ value for a PSP scheme, the better because that indicates that the scheme had a higher percentage of OD pairs with loss frequency less or equal to $x$. The figure shows that across the range of loss frequencies, CDF-PSP always has the highest percentage of OD pairs comparing to the other PSP schemes at any given $x$ value. In particular, both CDF-PSP and Mean-PSP substantially increase the number of OD pairs without packet loss at any of 48 attack time intervals, with CDF-PSP performing the best. The percentage of OD pairs with 0% loss frequency increase from 55.86% for No-PSP to 62.83% for Mean-PSP and 72.97% for CDF-PSP for the US network. The corresponding values for the EU network are 50.44%, 63.22% and 70.91%, respectively. In addition, for the US network, 98% of the OD pairs have loss frequencies bounded by 22.92% under Mean-PSP and 18.75% under CDF-PSP. Considering the 98% coverage of the OD pairs population under No-PSP, the bounding loss frequency is a much higher 66.67%. Thus, using either Mean-PSP or CDF-PSP substantially reduces the loss frequency for a large proportion of the crossfire OD pairs.

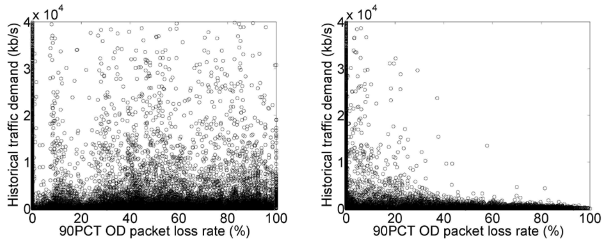
*2) Packet Loss Rate per OD pair:*

After exploring how often packet losses occur, we next analyze the magnitude of packet losses for different crossfire OD pairs. An OD-pair can have different loss rates at different attack time intervals, and here for each crossfire OD pair, we consider the $90^{th}$ percentile of these loss rates across time, where we consider only time intervals where that OD pair had non-zero traffic demand. Figure 13 shows the cumulative distribution function (CDF) of this $90^{th}$ ***percentile packet loss rate*** across all crossfire OD-pairs, except those that had no traffic demand during the entire 48 attack time intervals. In the figure, a given point $(x, y)$ indicates that for $y\%$ of crossfire OD-pairs, in 90% of the time intervals in which that OD pair had some traffic demand, the packet loss was at most $x\%$. The most interesting region from a practical performance perspective lies to the left of the graph for low values of the loss rate. This is because many network applications and even reliable transport protocols like TCP have very poor performance and are practically unusable beyond a loss rate of a few percentage points. Focussing on $0-10\%$ loss rate range which is widely considered to include this 'habitable zone of loss rates', the figure shows that both Mean-PSP and CDF-PSP both have substantially higher percentage of OD pairs in this zone, compared to No-PSP, and that CDF-PSP has significantly better performance. For example, the US network, the percentage of OD pair with less than 10% loss rate increases from just $59\%$ for No-PSP to 70.48% for Mean-PSP and 79.62% for CDF-PSP. The trends are similar for the EU network.

It should be noted that towards the tail of the distribution, for very large values of the loss rate, the percentage of OD pairs that have less than a certain loss rate $x$ is not always greater for CDF-PSP than for Mean-PSP. We defer the explanation for this to Section VI-C.4 where we analyze the packet losses of a OD-pair under different PSP schemes in greater detail.

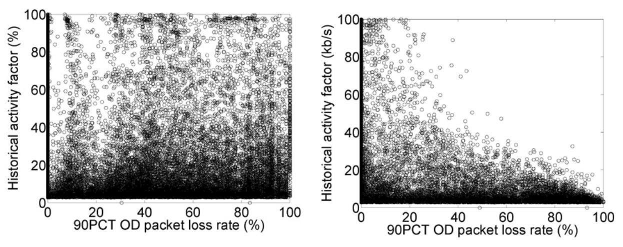
*3) Correlating Loss Rate with OD pair characteristics:*

The loss rate experienced by an OD pair for a PSP scheme is a function of various factors including the historical traffic demand for that OD pair which influences the admission decisions to the high priority class. To understand the relationship, we consider 2 simple features of its historical traffic profile. The ***historical traffic demand*** of an OD pair is the traffic demand for that OD pair averaged across all the historical time intervals. The ***historical activity factor*** is the percentage of time intervals that the OD pair had some traffic demand out of all historical time intervals. We explore the relation between each of these features and the $90^{th}$ percentile packet loss rate defined in the previous subsection in the



(a) US: No-PSP (b) US: CDF-PSP

Fig. 14. The correlation scatter plot for all crossfire OD-pairs between its 90 percentile OD packet loss rate under No-PSP/CDF-PSP and its historical traffic demand.



(a) US: No-PSP (b) US: CDF-PSP

Fig. 15. The correlation scatter plot for all crossfire OD-pairs between its 90 percentile OD packet loss rate under No-PSP/CDF-PSP and its historical activity factor.

scatter plots in Figures 14[2] and 15[3], where each dot corresponds to a crossfire OD pair and the location of the dot is determined by its $90^{th}$ percentile packet loss rate and either its historical demand (Figure 14) or its historical activity factor (Figure 15).

Comparing the results for No-PSP and CDF-PSP in the 2 figures, we note that unlike No-PSP, under CDF-PSP, the top right region in the plots are empty and that no OD pair with high historical demand or high historical activity has a high loss rate. Since the historical demand and activity factor values for an OD pair does not change from No-PSP to CDF-PSP, the scatter plots indicate that for many high demand or high activity factor OD pairs, the loss rates are dramatically reduced going from No-PSP to CDF-PSP, shifting their corresponding points to the left side. Under CDF-PSP, all the points with high loss rates correspond to OD pairs with low historical demand or activity factors.

This suggests that CDF-PSP provides better protection for OD pairs with high demand or high activity. This is very desirable from a service provider perspective because OD pairs with high demand or high activity typically carry traffic from large customers who pay the most and are the most sensitive to service interruptions.

*4) OD pair Loss Improvement:*

As mentioned in Section VI-C.2, CDF-PSP does not always result in a lower packet loss for every OD pair than Mean-PSP. This can be attributed to the different amounts of packets being marked in the high priority class for an OD pair under different policies. It is also possible that both PSP techniques may exhibit higher loss rates for some OD pair in some time interval, compared to No-PSP. This is because under either PSP scheme, under high load conditions, most of the network capacity is used to serve high priority packets, and any residual capacity is used to serve low priority packets. Therefore packets that are marked as low priority will tend to have higher drop rates than under No-PSP, where all packets were treated equally. Therefore for an OD pair, if a large proportion of its offered load gets marked as low priority, and there is congestion on the path, in theory it could suffer more losses than under No-PSP. However, this should not be a common case, since the PSP bandwidth allocation is designed to accommodate the normal traffic demand of an OD pair in the high priority class, based on historical demands. In the following, we examine how often CDF-PSP has better performance than either No-PSP or Mean-PSP.

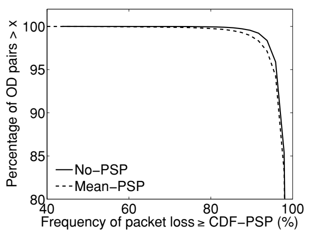
For both No-PSP and Mean-PSP, we determine for each OD pair the percentage of the 48 attack time intervals when the packet loss rate was no less than the loss rate under CDF-PSP. We plot the complementary cumulative distribution function (CCDF) of this value across all crossfire OD pairs with demand at any of the 48 attack time intervals, for No-PSP and Mean-PSP in Figure 16. For each curve, a given point $(x, y)$ in the figure indicates that for $y$ percent of the crossfire OD pairs, the loss rates are greater than or equal to that under CDF-PSP in at least $x$ percent of the time intervals. The graphs indicate that CDF-PSP outperforms both No-PSP and Mean-PSP for most OD pairs in a large proportion of the time intervals. Compared to No-PSP, for the EU network, under CDF-PSP, $90.72\%$ of the OD pairs have equal or lower loss rates in all 48 time intervals, and $98\%$ of the OD pairs have lower loss rates in at least $93.75\%$ of the time intervals. For the same network, compared to Mean-PSP, CDF-PSP resulted in equal or lower loss rates at all 48 time intervals for $81.27\%$ of the OD pairs.

## *D. Performance under scaled attacks*

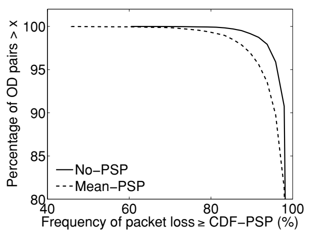
Given the growing penetration of broadband connections and the ever-increasing availability of large armies of botnets "for hire", it is important to understand the effectiveness of the PSP techniques with respect to increasing attack intensity. To study this, for each network, we vary the intensity of the attack matrix by scaling the

[2]The y-axis is cut off at 40,000 kb/s because only a few OD pairs exceeded that demand and all of them had less than 10% loss rate.

[3]Due to space constraints, we only show the results for the US network, while the results are similar in the EU network.

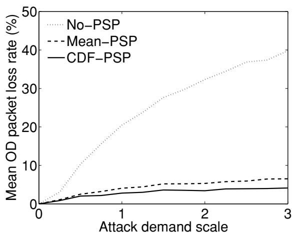


(a) US

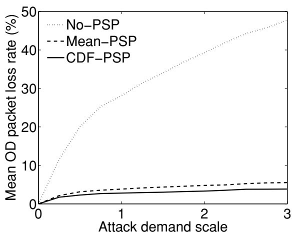


(b) EU

Fig. 16. CCDF of percentage of time that the loss rate for a crossfire OD pair under No-PSP and Mean-PSP exceeds that under CDF-PSP



(a) US



(b) EU

Fig. 17. The time-averaged mean crossfire OD-pair packet loss rate as the attack volume scaling factor increases from 0 to 3.

demand of every attack flow by a factor ranging from 0 to 3, in steps of size 0.25. For each value of the scaling factor, we measure the time-averaged Mean OD packet loss rate of crossfire OD pairs (defined in Section VI-B.2) across eight 1-min. time intervals, equally spaces across 24 hours. Figure 17 shows that the loss rate under No-PSP increases much faster than under Mean-PSP and CDF-PSP, as the attack intensity increases. This is because under No-PSP, all the normal traffic packets have to compete for limited bandwidth resources with the attack traffic, while with our protection scheme only normal traffic marked in low priority class is affected by the increasing attack. Therefore, even in the extreme case when the attack traffic demand is sufficient to clog all links, our protection scheme can still guarantee that the normal traffic marked in the high priority class goes through the network. Consequently, our PSP schemes are much less sensitive to the degree of congestion, as evident by the much slower growth of the drop rate. For example, in the US network, as the scale factor increases from 1 to 3, under No-PSP, the mean drop rate jumped from slightly above $20\%$ to almost $40\%$ . In comparison, under CDF-PSP, the mean loss rate increases very little from less than $3\%$ to $4\%$ over the same range of attack intensities. The trends demonstrate that across the range of scaling factor values, both the PSP schemes are very effective in mitigating collateral damage by keeping loss rates low, with CDF-PSP having an edge over Mean-PSP.

### *E. Summary of Results*

In this section, we summarize the main findings from the evaluation of our PSP methods on two large backbone networks. First, we show that the potential for collateral damage is significant in that even when a small number of OD pairs are attacked, a majority of the OD pairs in a network can be substantially impacted. For both the US and EU backbones, we observed that the percentage of OD pairs impacted is surprisingly large, 95.5% and 83.5%, even though the attacks were directed over only 1.2% and 0.1% of the OD pairs, respectively. Comparing to no protection, Mean-PSP and CDF-PSP significantly reduced the total packet loss up to 97.58%, the mean OD pair packet loss rates up to 95.92%, and the number of crossfire OD pairs with packet loss by 90.36%. Further, CDF-PSP substantially improved over Mean-PSP by reducing the loss rate across all evaluation matrices. Specifically, CDF-PSP reduced the total packet loss of Mean-PSP up to 53.09% in the US network and up to 41.58% in the EU network, and CDF-PSP reduced the number of OD pairs with packet loss by up to 59.30% in the US network and up to 47.60% in the EU network. Finally, we show PSP can maintain low packet loss rates even when the intensity of attacks is increased significantly.

## VII. Conclusion

PSP provides network operators with a broad first line of proactive defense against DDoS attacks, significantly reducing the impact of sudden bandwidth-based attacks on a service provider network. The proactive surge protection is achieved by providing bandwidth isolation between traffic flows. This isolation is achieved through a combination of traffic data collection, bandwidth allocation of network resources, metering and tagging of packets at the network perimeter, and preferential dropping of packets inside the network. Among its salient features, PSP is readily deployable using existing router mechanisms, and PSP does not rely on any unauthenticated packet header information. The latter feature makes the solution resilient to evading attack schemes that launch many seemingly legitimate TCP connections with spoofed IP addresses and port numbers. This is due to the fact that PSP focuses on protecting traffic between different ingress-egress interface pairs in a provider network, and both the ingress and egress interface of an IP datagram can be directly determined by the network operator. By taking into consideration traffic variability observed in traffic measurements, our proactive protection solution can ensure the maximization of the acceptance probability of each flow in a max-min fair manner, or equivalently the minimization of the drop probability in a min-max fair manner. Our extensive evaluation results across two large commercial backbone networks, using both distributed and targeted attack scenarios, show that up to 95.5% of the network could suffer collateral damage without protection, but our solution was able to significantly reduce the amount of collateral damage by up to 97.58% in terms of the number of packets dropped and 90.36% in terms of the number of flows with packet loss. In addition, we show that PSP can maintain low packet loss rates even when the intensity of attacks is increased significantly.

## REFERENCES

[1] Advanced networking for leading-edge research and education. http://abilene.internet2.edu.

[2] Arbor peakflow. www.arbor.net.

[3] CERT CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks.

[4] Cisco guard. http://www.cisco.com/en/US/products/ps5888/index.html.

[5] Distributed weighted random early detection. http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.pdf.

[6] Washington Post, The Botnet Trackers, Tursday, February 16, 2006.

[7] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *ACM HotNets Workshop*, November 2005.

[8] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[9] Z. Cao and E. W. Zegura. Utility max-min: An application-oriented bandwidth allocation scheme. In *IEEE INFOCOM*, pages 793–801, 1999.

[10] J. Chou, B. Lin, S. Sen, and O. Spatscheck. Minimizing collateral damage by Proactive Surge Protection. In *ACM LSAD Workshop*, pages 97–104, August 2007.

[11] D. Clark and W. Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM ToN*, August 1998.

[12] N. G. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM*, August 2003.

[13] M. A. El-Gendy, A. Bose, and K. G. Shin. Evolution of the Internet QoS and support for soft real-time applications. *Proceedings of the IEEE*, 91(7):1086–1104, July 2003.

[14] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. In *ACM SIGCOMM*, June 2000.

[15] M. Grossglauser and D. N. C. Tse. A framework for robust measurement-based admission control. In *IEEE/ACM ToN*, 1999.

[16] Y. Hou, H. Tzeng, and S. Panwar. A generalized max-min rate allocation policy and its distributed implementationusing the ABR flow control mechanism. In *IEEE INFOCOM*, pages 1366–1375, 1998.

[17] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *Network and Distributed System Security Symposium*, 1775 Wiehle Ave., Suite 102, Reston, VA 20190, February 2002. The Internet Society.

[18] S. Jamin, P. B. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. *IEEE/ACM ToN*, February 1996.

[19] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds. In *ACM/USENIX NSDI*, May 2005.

[20] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Taming IP packet flooding attacks. In *ACM HotNets Workshop*, 2003.

[21] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina. Detection and identification of network anomalies using sketch subspaces. In *ACM/USENIX IMC*, October 2006.

[22] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *International Conference on Network Protocols*, November 2001.

[23] B. Radunovic and J.-Y. L. Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM ToN*, accepted for publication.

[24] B. Raghavan and A. C. Snoeren. A system for authenticated policy-compliant routing. In *ACM SIGCOMM*, October 2004.

[25] J. Ros and W. Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *IEEE INFOCOM*, pages 717–726, 2001.

[26] D. Rubenstein, J. Kurose, and D. Towsley. The impact of multicast layering on network fairness. *IEEE/ACM ToN*, April 2002.

[27] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Network support for IP traceback. *IEEE/ACM ToN*, 9(3), June 2001.

[28] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM ToN*, 10(6):721–734, December 2002.

[29] H. Tzeng and K. Siu. On max-min fair congestion control for multicast ABR service in ATM. *IEEE Journal on Selected Areas in Communications*, 1997.

[30] P. Verkaik, O. Spatscheck, J. V. der Merwe, and A. C. Snoeren. Primed: community-of-interest-based ddos mitigation. In *ACM LSAD Workshop*, pages 147–154, November 2006.

[31] Y. Xu and R. Guérin. On the robustness of router-based denial-of-service (dos) defense systems. *SIGCOMM Comput. Commun. Rev.*, 35(3):47–60, 2005.

[32] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *IEEE Security and Privacy Symposium*, pages 93–107, May 2003.

[33] A. Yaar, A. Perrig, and D. Song. An endhost capability mechanism to mitigate DDoS flooding attacks. In *IEEE Security and Privacy Symposium*, May 2004.

[34] D. K. Y. Yau, J. C. S. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM ToN*, 13(1):29–42, 2005.

# BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection

*Guofei Gu*[†]*, Roberto Perdisci*[‡]*, Junjie Zhang*[†]*, and Wenke Lee*[†]
[†]*College of Computing, Georgia Institute of Technology*
[‡]*Damballa, Inc. Atlanta, GA 30308, USA*
{`guofei,jjzhang,wenke`}`@cc.gatech.edu, perdisci@damballa.com`

## Abstract

Botnets are now the key platform for many Internet attacks, such as spam, distributed denial-of-service (DDoS), identity theft, and phishing. Most of the current botnet detection approaches work only on specific botnet command and control (C&C) protocols (e.g., `IRC`) and structures (e.g., centralized), and can become ineffective as botnets change their C&C techniques. In this paper, we present a general detection framework that is independent of botnet C&C protocol and structure, and requires no *a priori* knowledge of botnets (such as captured bot binaries and hence the botnet signatures, and C&C server names/addresses). We start from the definition and essential properties of botnets. We define a botnet as a *coordinated group* of *malware* instances that are *controlled* via C&C communication channels. The essential properties of a botnet are that the bots communicate with some C&C servers/peers, perform malicious activities, and do so in a similar or correlated way. Accordingly, our detection framework clusters similar communication traffic and similar malicious traffic, and performs cross cluster correlation to identify the hosts that share both similar communication patterns *and* similar malicious activity patterns. These hosts are thus bots in the monitored network. We have implemented our BotMiner prototype system and evaluated it using many real network traces. The results show that it can detect real-world botnets (`IRC`-based, `HTTP`-based, and P2P botnets including Nugache and Storm worm), and has a very low false positive rate.

## 1 Introduction

Botnets are becoming one of the most serious threats to Internet security. A botnet is a network of compromised machines under the influence of malware (bot) code. The botnet is commandeered by a "botmaster" and utilized as "resource" or "platform" for attacks such as distributed denial-of-service (DDoS) attacks, and fraudulent activities such as spam, phishing, identity theft, and information exfiltration.

In order for a botmaster to command a botnet, there needs to be a command and control (C&C) channel through which bots receive commands and coordinate attacks and fraudulent activities. The C&C channel is the means by which individual bots form a bot*net*. Centralized C&C structures using the Internet Relay Chat (`IRC`) protocol have been utilized by botmasters for a long time. In this architecture, each bot logs into an `IRC` channel, and seeks commands from the botmaster. Even today, many botnets are still designed this way. Quite a few botnets, though, have begun to use other protocols such as `HTTP` [8, 14, 24, 39], probably because `HTTP`-based C&C communications are more stealthy given that Web traffic is generally allowed in most networks. Although centralized C&C structures are effective, they suffer from the single-point-of-failure problem. For example, if the `IRC` channel (or the Web server) is taken down due to detection and response efforts, the botnet loses its C&C structure and becomes a collection of isolated compromised machines. Recently, botmasters began using peer-to-peer (P2P) communication to avoid this weakness. For example, Nugache [28] and Storm worm [18, 23] (a.k.a. Peacomm) are two representative P2P botnets. Storm, in particular, distinguishes itself as having infected a large number of computers on the Internet and effectively becoming one of the "world's top super-computers" [27] for the botmasters.

Researchers have proposed a few approaches [7, 17, 19, 20, 26, 29, 35, 40] to detect the existence of botnets in monitored networks. Almost all of these approaches are designed for detecting botnets that use `IRC` or `HTTP` based C&C [7, 17, 26, 29, 40]. For example, Rishi [17] is designed to detect `IRC` botnets using known IRC bot nickname patterns as signatures. In [26, 40], network flows are clustered and classified according to IRC-like traffic patterns. Another more recent system, BotSniffer, [20] is designed mainly for detecting C&C activities with centralized servers (with protocols such as `IRC` and `HTTP`[1]). One exception is perhaps BotHunter [19], which is capable of detecting bots regardless of the C&C structure and network protocol as long as the bot behavior follows a *pre-defined* infection life cycle dialog model.

However, botnets are evolving and can be quite flexible. We have witnessed that the protocols used for C&C evolved from `IRC` to others (e.g., `HTTP` [8, 14, 24, 39]), and the structure moved from centralized to distributed (e.g., using P2P [18, 28]). Furthermore, a botnet during its lifetime can also change its C&C server address frequently, e.g., using fast-flux service networks [22]. Thus, the aforementioned detection approaches designed for `IRC` or `HTTP` based botnets may become ineffective against the recent/new botnets. Even BotHunter may fail as soon as botnets change their infection model(s).

Therefore, we need to develop a next generation botnet detection system, which should be independent of the C&C protocol, structure, and infection model of botnets, and be resilient to the change of C&C server addresses. In addition, it should require no *a priori* knowledge of specific botnets (such as captured bot binaries and hence the botnet signatures, and C&C server names/addresses).

In order to design such a general detection system that can resist evolution and changes in botnet C&C techniques, we need to study the *intrinsic* botnet communication and activity characteristics that remain detectable with the proper detection features and algorithms. We thus start with the definition and essential properties of a botnet. We define a botnet as:

"A *coordinated group* of *malware* instances that are *controlled* via C&C channels".

The term "malware" means these bots are used to perform *malicious activities*. According to [44], about 53% of botnet activity commands observed in thousands of real-world `IRC`-based botnets are related to scan (for the purpose of spreading or DDoS[2]), and about 14.4% are related to binary downloading (for the purpose of malware updating). In addition, most of `HTTP`-based and P2P-based botnets are used to send spam [18, 39]. The term "controlled" means these bots have to contact their C&C servers to obtain commands to carry out activities, e.g., to scan. In other words, there should be *communication between bots and C&C servers/peers* (which can be centralized or distributed). Finally, the term "coordinated group" means that multiple (at least two) bots within the same botnet will perform *similar* or *correlated* C&C communications and malicious activities. If the botmaster commands each bot individually with a different command/channel, the bots are nothing but some isolated/unrelated infections. That is, they do not function as a bot*net* according to our definition and are out of the scope of this work[3].

We propose a general detection framework that is based on these essential properties of botnets. This framework monitors both *who is talking to whom* that may suggest C&C communication activities and *who is doing what* that may suggest malicious activities, and finds a *coordinated group pattern* in both kinds of activities. More specifically, our detection framework clusters similar communication activities in the *C-plane* (C&C communication traffic), clusters similar malicious activities in the *A-plane* (activity traffic), and performs cross cluster correlation to identify the hosts that share both similar communication patterns *and* similar malicious activity patterns. These hosts, according to the botnet definition and properties discussed above, are bots in the monitored network.

This paper makes the following main contributions.

- We develop a novel general botnet detection framework that is grounded on the definition and essential properties of botnets. Our detection framework is thus independent of botnet C&C protocol and structure, and requires no *a priori* knowledge (e.g., C&C addresses/signatures) of specific botnets. It can detect both centralized (e.g., `IRC,HTTP`) and current (and possibly future) P2P based botnets.

- We define a new "aggregated communication flow" (C-flow) record data structure to store aggregated traffic statistics, and design a new layered clustering scheme with a set of traffic features measured on the C-flow records. Our clustering scheme can accurately and efficiently group similar C&C traffic patterns.

- We build a BotMiner prototype system based on our general detection framework, and evaluate it with multiple real-world network traces including normal traffic and several real-world botnet traces that contain `IRC`, `HTTP` and P2P-based botnet traffic (including Nugache and Storm). The results show that BotMiner has a high detection rate and a low false positive rate.

The rest of the paper is organized as follows. In Section 2, we describe the assumptions, objectives, architecture of our BotMiner detection framework, and its

[1]BotSniffer could be extended to support other protocol based C&C, if the corresponding protocol matchers are added.

[2]For spreading, the scans usually span many different hosts (within a subnet) indicated by the botnet command. For DDoS, usually there are numerous connection attempts to a specific host. In both cases, the traffic can be considered as scanning related.

[3]One can still use our complementary system, BotHunter [19], to detect individual bots. In this paper, we focus on the detection of a bot*net*. We further clarify our assumptions in Section 2.1 and address limitations in Section 4.

detection algorithms and implementation. In Section 3, we describe our evaluation on various real-world network traces. In Section 4, we discuss current limitations and possible solutions. We review the related work in Section 5 and conclude in Section 6.

## 2 BotMiner Detection Framework and Implementation

### 2.1 Problem Statement and Assumptions

According to the definition given above, a botnet is characterized by both a C&C communication channel (from which the botmaster's commands are received) and malicious activities (when commands are executed). Some other forms of malware (e.g., worms) may perform malicious activities, but they do not connect to a C&C channel. On the other hand, some normal applications (e.g., `IRC` clients and normal P2P file sharing software) may show communication patterns similar to a botnet's C&C channel, but they do not perform malicious activities.

Figure 1 illustrates two typical botnet structures, namely *centralized* and *P2P*. The bots receive commands from the botmaster using a *push* or *pull* mechanism [20] and execute the assigned tasks.

The operation of a centralized botnet is relatively easy and intuitive [20], whereas this is not necessarily true for P2P botnets. Therefore, here we briefly illustrate an example of a typical P2P-based botnet, namely Storm worm [18, 23]. In order to issue commands to the bots, the botmaster publishes/shares command files over the P2P network, along with specific search keys that can be used by the bots to find the published command files. Storm bots utilize a pull mechanism to receive the commands. Specifically, each bot frequently contacts its neighbor peers searching for specific keys in order to locate the related command files. In addition to search operations, the bots also frequently communicate with their peers and send *keep-alive* messages.

In both centralized and P2P structures, bots within the same botnet are likely to behave similarly in terms of communication patterns. This is largely due to the fact that bots are non-human driven, pre-programmed to perform the same routine C&C logic/communication as coordinated by the same botmaster. In the centralized structure, even if the address of the C&C server may change frequently (e.g., by frequently changing the A record of a Dynamic DNS domain name), the C&C communication patterns remain unchanged. In the case of P2P-based botnets, the peer communications (e.g., to search for commands or to send *keep-alive* messages) follow a similar pattern for all the bots in the botnet, although each bot may have a different set of neighbor peers and may communicate on different ports.

Regardless of the specific structure of the botnet (centralized or P2P), members of the same botnet (i.e., the bots) are coordinated through the C&C channel. In general, a bot*net* is different from a set of *isolated* individual malware instances, in which each different instance is used for a totally different purpose. Although in an extreme case a botnet can be configured to degenerate into a group of *isolated hosts*, this is not the common case. In this paper, we focus on the most typical and useful situation in which bots in the same bot*net* perform similar/coordinated activities. To the best of our knowledge, this holds true for most of the existing botnets observed in the wild.

To summarize, we assume that bots within the same botnet will be characterized by similar malicious activities, as well as similar C&C communication patterns. Our assumption holds even in the case when the botmaster chooses to divide a botnet into *sub-botnets*, for example by assigning different tasks to different sets of bots. In this case, each sub-botnet will be characterized by similar malicious activities and C&C communications patterns, and our goal is to detect each sub-botnet. In Section 4 we provide a detailed discussion on possible *evasive* botnets that may violate our assumptions.

### 2.2 Objectives

The objective of BotMiner is to detect *groups* of compromised machines within a monitored network that are part of a botnet. We do so by passively analyzing network traffic in the monitored network.

Note that we do not aim to detect botnets at the very moment when victim machines are compromised and infected with malware (bot) code. In many cases these events may not be observable by passively monitoring network traffic. For example, an already infected laptop may be carried in and connected to the monitored network, or a user may click on a malicious email attachment and get infected. In this paper we are not concerned with the way internal hosts become infected (e.g., by malicious email attachments, remote exploiting, and Web drive-by download). We focus on the detection of groups of already compromised machines inside the monitored network that are part of a botnet.

Our detection approach meets several goals:

- it is independent of the protocol and structure used for communicating with the botmaster (the C&C channel) or peers, and is resistant to changes in the location of the C&C server(s).

- it is independent of the content of the C&C communication. That is, we do not inspect the content of the C&C communication itself, because C&C could be encrypted or use a customized (obscure) protocol.

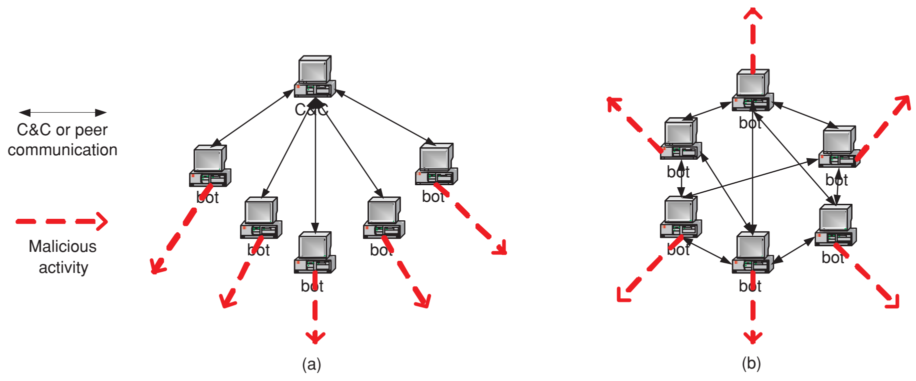


Figure 1: Possible structures of a botnet: (a) centralized; (b) peer-to-peer.

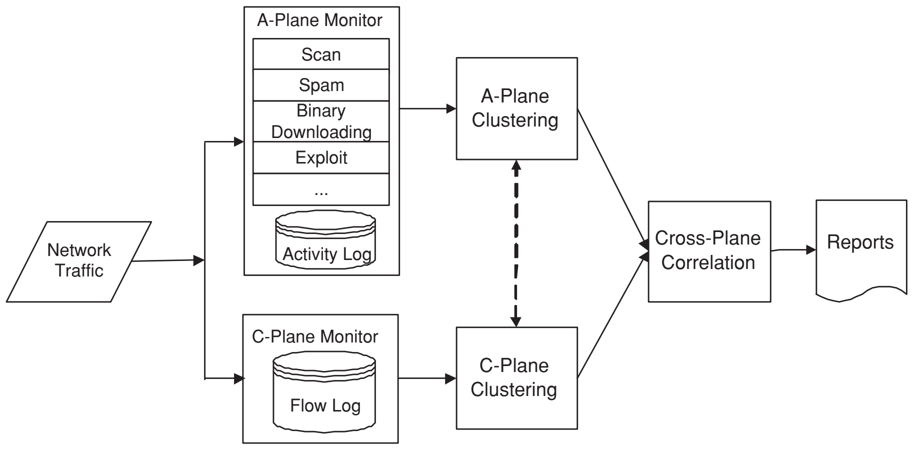


Figure 2: Architecture overview of our BotMiner detection framework.

- it generates a low number of false positives and false negatives.

- the analysis of network traffic employs a reasonable amount of resources and time, making detection relatively efficient.

## 2.3 Architecture of BotMiner Detection Framework

Figure 2 shows the architecture of our BotMiner detection system, which consists of five main components: C-plane monitor, A-plane monitor, C-plane clustering module, A-plane clustering module, and cross-plane correlator.

The two traffic monitors in C-plane and A-plane can be deployed at the edge of the network examining traffic between internal and external networks, similar to BotHunter [19] and BotSniffer [20][4]. They run in parallel and monitor the network traffic. The C-plane monitor is responsible for logging network flows in a format suitable for efficient storage and further analysis, and the A-plane monitor is responsible for detecting suspicious activities (e.g., scanning, spamming, and exploit attempts). The C-plane clustering and A-plane clustering components process the logs generated by the C-plane and A-plane monitors, respectively. Both modules extract a number of features from the raw logs and apply clustering algorithms in order to find groups of machines that show very similar communication (in the C-plane) and activity (in the A-plane) patterns. Finally, the cross-plane correlator combines the results of the C-plane and A-plane clustering and makes a final decision on which machines are possibly members of a botnet. In an ideal situation, the traffic monitors should be distributed on the Internet, and the monitor logs are reported to a central repository for clustering and cross-plane analysis.

[4]All these tools can also be deployed in LANs.

In our current prototype system, traffic monitors are implemented in C for the purpose of efficiency (working on real-time network traffic). The clustering and correlation analysis components are implemented mainly in Java and R (`http://www.r-project.org/`), and they work offline on logs generated from the monitors.

The following sections present the details of the design and implementation of each component of the detection framework.

### 2.4 Traffic Monitors

**C-plane Monitor.** The C-plane monitor captures network flows and records information on *who is talking to whom*. Many network routers support the logging of network flows, e.g., Cisco (`www.cisco.com`) and Juniper (`www.juniper.net`) routers. Open source solutions like Argus (Audit Record Generation and Utilization System, `http://www.qosient.com/argus`) are also available. We adapted an efficient network flow capture tool developed at our research lab, i.e., `fcapture` [5], which is based on the Judy library (`http://judy.sourceforge.net/`). Currently, we limit our interest to TCP and UDP flows. Each flow record contains the following information: time, duration, source IP, source port, destination IP, destination port, and the number of packets and bytes transfered in both directions. The main advantage of our tool is that it works very efficiently on high speed networks (very low packet loss ratio on a network with 300Mbps traffic), and can generate very compact flow records that comply with the requirement for further processing by the C-plain clustering module. As a comparison, our flow capturing tool generates compressed records ranging from 200MB to 1GB per day from the traffic in our academic network, whereas Argus generates around 36GB of compressed binary flow records per day on average (without recording any payload information). Our tool makes the storage of several weeks or even months of flow data feasible.

**A-plane Monitor.** The A-plane monitor logs information on *who is doing what*. It analyzes the outbound traffic through the monitored network and is capable of detecting several malicious activities that the internal hosts may perform. For example, the A-plane monitor is able to detect scanning activities (which may be used for malware propagation or DoS attacks), spamming, binary downloading (possibly used for malware update), and exploit attempts (used for malware propagation or targeted attacks). These are the most common and "useful" activities a botmaster may command his bots to perform [9, 33, 44].

Our A-plane monitor is built based on Snort [36], an open-source intrusion detection tool, for the purpose of convenience. We adapted existing intrusion detection techniques and implemented them as Snort pre-processor plug-ins or signatures. For scan detection we adapted SCADE (Statistical sCan Anomaly Detection Engine), which is a part of BotHunter [19] and available at [11]. Specifically, we mainly use two anomaly detection modules: the *abnormally-high scan rate* and weighted *failed connection rate*. We use an `OR` combination rule, so that an event detected by either of the two modules will trigger an alert. In order to detect spam-related activities, we developed a new Snort plug-in. We focused on detecting anomalous amounts of DNS queries for MX records from the same source IP and the amount of SMTP connections initiated by the same source to mail servers outside the monitored network. Normal clients are unlikely to act as SMTP servers and therefore should rely on the internal SMTP server for sending emails. Use of many distinct external SMTP servers for many times by the same internal host is an indication of possible malicious activities. For the detection of PE (Portable Executable) binary downloading we used an approach similar to PEHunter [42] and BotHunter's egg download detection method [19]. One can also use specific exploit rules in BotHunter to detect internal hosts that attempt to exploit external machines. Other state-of-the-art detection techniques can be easily added to our A-plane monitoring to expand its ability to detect typical botnet-related malicious activities.

It is important to note that A-plane monitoring alone is not sufficient for botnet detection purpose. First of all, these A-plane activities are not exclusively used in botnets. Second, because of our relatively loose design of A-plane monitor (for example, we will generate a log whenever there is a PE binary downloading in the network regardless of whether the binary is malicious or not), relying on only the logs from these activities will generate a lot of false positives. This is why we need to further perform A-plane clustering analysis as discussed shortly in Section 2.6.

### 2.5 C-plane Clustering

C-plane clustering is responsible for reading the logs generated by the C-plane monitor and finding clusters of machines that share similar communication patterns. Figure 3 shows the architecture of the C-plane clustering.

First of all, we filter out irrelevant (or uninteresting) traffic flows. This is done in two steps: basic-filtering and white-listing. It is worth noting that these two steps are not critical for the proper functioning of the C-plane clustering module. Nonetheless, they are useful for reducing the traffic workload and making the actual clustering process more efficient. In the basic-filtering step, we filter out all the flows that are not directed from internal hosts to external hosts. Therefore, we ignore the flows related to communications between internal hosts[6] and flows initiated from external hosts towards internal hosts (filter rule 1, denoted as `F1`). We also filter out flows that are not completely established (filter rule 2,

[5]This tool will be released in open source soon.

[6]If the C-plane monitor is deployed at the edge router, these traffic will not be seen. However, if the monitor is deployed/tested in a LAN, then this filtering can be used.

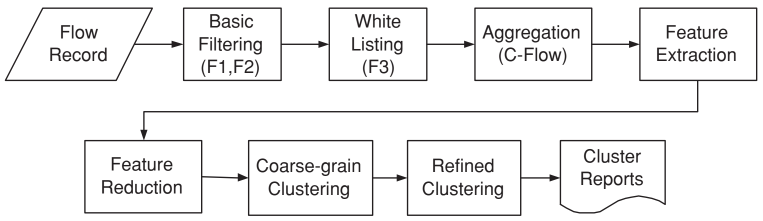


Figure 3: C-plane clustering.

denoted as F2), i.e., those flows that only contain one-way traffic. These flows are mainly caused by scanning activity (e.g., when a host sends SYN packets without completing the TCP hand-shake). In white-list filtering, we filter out those flows whose destinations are well known as legitimate servers (e.g., Google, Yahoo!) that will unlikely host botnet C&C servers. This filter rule is denoted as F3. In our current evaluation, the white list is based on the US top 100 and global top 100 most popular websites from Alexa.com.

After basic-filtering and white-listing, we further reduce the traffic workload by aggregating related flows into communication flows (C-flows) as follows. Given an epoch $E$ (typically one day), all $m$ TCP/UDP flows that share the same protocol (TCP or UDP), source IP, destination IP and port, are aggregated into the same C-flow $c_i = \{f_j\}_{j=1..m}$, where each $f_j$ is a single TCP/UDP flow. Basically, the set $\{c_i\}_{i=1..n}$ of all the $n$ C-flows observed during $E$ tells us "who was talking to whom", during that epoch.

### 2.5.1 Vector Representation of C-flows

The objective of C-plane clustering is to group hosts that share similar communication flows. This can be accomplished by clustering the C-flows. In order to apply clustering algorithms to C-flows we first need to translate them in a suitable vector representation. We extract a number of statistical features from each C-flow $c_i$, and translate them into $d$-dimensional pattern vectors $\vec{p_i} \in \mathbb{R}^d$. We can describe this task as a projection function $F$ : C-plane $\rightarrow \mathbb{R}^d$. The projection function $F$ is defined as follows. Given a C-flow $c_i$, we compute the discrete sample distribution of (currently) four random variables:

1. *the number of flows per hour (fph). fph* is computed by counting the number of TCP/IP flows in $c_i$ that are present for each hour of the epoch $E$.

2. *the number of packets per flow (ppf). ppf* is computed by summing the total number of packets sent within each TCP/UDP flow in $c_i$.

3. *the average number of bytes per packets (bpp).* For each TCP/UDP flow $f_j \in c_i$ we divide the overall number of bytes transfered within $f_j$ by the number of packets sent within $f_j$.

4. *the average number of bytes per second (bps). bps* is computed as the total number of bytes transfered within each $f_j \in c_i$ divided by the duration of $f_j$.

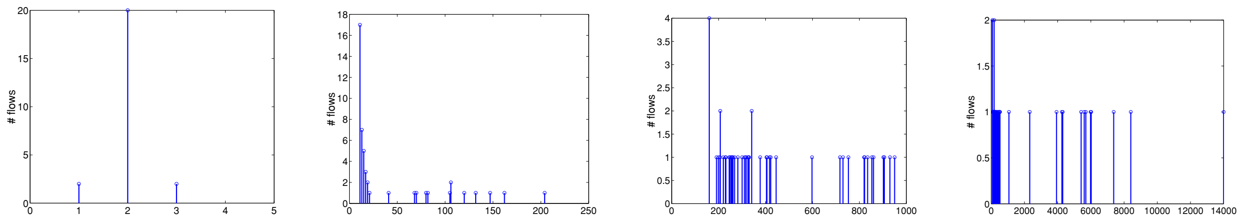
An example of the results of this process is shown in Figure 4, where we select a random client from a real network flow log (we consider a one-day epoch) and illustrate the features extracted from its visits to Google.

Given the discrete sample distribution of each of these four random variables, we compute an approximate version of it by means of a binning technique. For example, in order to approximate the distribution of *fph* we divide the x-axis in 13 intervals as $[0, k_1], (k_1, k_2], ..., (k_{12}, \infty)$. The values $k_1, .., k_{12}$ are computed as follows. First, we compute the overall discrete sample distribution of $fph$ considering all the C-flows in the traffic for an epoch $E$. Then, we compute the quantiles[7] $q_{5\%}, q_{10\%}, q_{15\%}, q_{20\%}, q_{25\%}, q_{30\%}, q_{40\%}, q_{50\%}, q_{60\%}, q_{70\%}, q_{80\%}, q_{90\%}$, of the obtained distribution, and we set $k_1 = q_{5\%}$, $k_2 = q_{10\%}$, $k_3 = q_{15\%}$, etc. Now, for each C-flow we can describe its *fph* (approximate) distribution as a vector of 13 elements, where each element $i$ represents the number of times *fph* assumed a value within the corresponding interval $(k_{i-1}, k_i]$. We also apply the same algorithm for *ppf*, *bpp*, and *bps*, and therefore we map each C-flow $c_i$ into a pattern vector $\vec{p_i}$ of $d = 52$ elements. Figure 5 shows the scaled visiting pattern extracted form the same C-flow shown in Figure 4.

### 2.5.2 Two-step Clustering

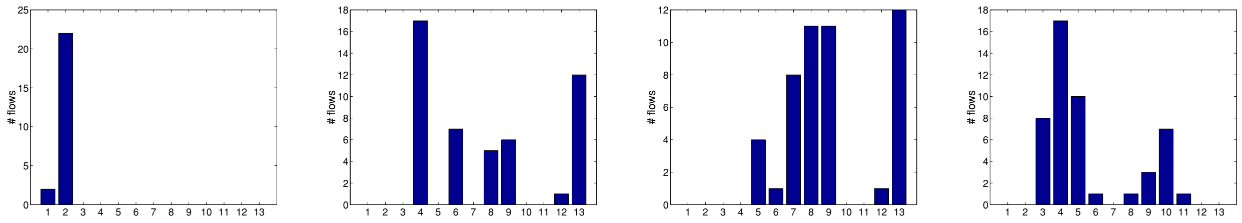
Since bots belonging to the same botnet share similar behavior (from both the communication and activity points of view) as we discussed before, our objective is to look for groups of C-flows that are similar to each other. Intuitively, pattern vectors that are close to each other in $\mathbb{R}^d$ represent C-flows with similar communication patterns in the C-plane. For example, suppose two bots of the same botnet connect to two different

[7]The quantile $q_{l\%}$ of a random variable $X$ is the value $q$ for which $P(X < q) = l\%$.



(a) fph (b) ppf (c) bpp (d) bps

Figure 4: Visit pattern (shown in distribution) to `Google` from a randomly chosen normal client.



(a) Scaled fph (b) Scaled ppf (c) Scaled bpp (d) Scaled bps

Figure 5: Scaled visit pattern (shown in distribution) to `Google` for the same client in Figure 4.

C&C servers (because some botnets use multiple C&C servers). Although the connections from both bots to the C&C servers will be in different C-flows because of different source/destination pairs, their C&C traffic characteristics should be similar. That is, in $\mathbb{R}^d$, these C-flows should be found as being very similar. In order to find groups of hosts that share similar communication patterns, we apply clustering techniques on the dataset $\mathcal{D} = \{\vec{p}_i = F(c_i)\}_{i=1..n}$ of the pattern vector representations of C-flows. Clustering techniques perform unsupervised learning. Typically, they aim at finding meaningful groups of data points in a given feature space $\mathbb{F}$. The definition of "meaningful clusters" is application-dependent. Generally speaking, the goal is to group the data into clusters that are both compact and well separated from each other, according to a suitable similarity metric defined in the feature space $\mathbb{F}$ [25].

Clustering C-flows is a challenging task because $|\mathcal{D}|$, the cardinality of $\mathcal{D}$, is often large even for moderately large networks, and the dimensionality $d$ of the feature space is also large. Furthermore, because the percentage of machines in a network that are infected by bots is generally small, we need to separate the few botnet-related C-flows from a large number of benign C-flows. All these make clustering of C-flows very expensive.

In order to cope with the complexity of clustering of $\mathcal{D}$, we solve the problem in several steps (currently in two steps), as shown in a simple form in Figure 6. At the first step, we perform coarse-grained clustering on a reduced feature space $\mathbb{R}^{d'}$, with $d' < d$, using a simple (i.e., non-expensive) clustering algorithm (we will explain below how we perform dimensionality reduction). The results

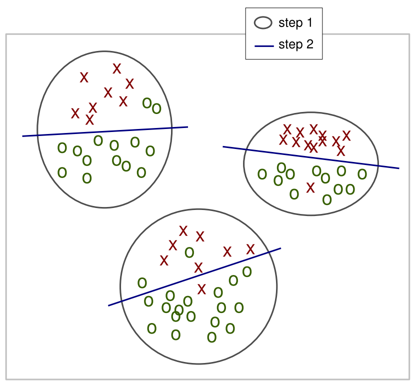


Figure 6: Two-step clustering of C-flows.

of this first-step clustering is a set $\{\mathcal{C}'_i\}_{i=1..\gamma_1}$ of $\gamma_1$ relatively large clusters. By doing so we subdivide the dataset $\mathcal{D}$ into smaller datasets (the clusters $\mathcal{C}'_i$) that contain "clouds" of points that are not too far from each other.

Afterwards, we refine this result by performing a second-step clustering on each different dataset $\mathcal{C}'_i$ using a simple clustering algorithm on the complete description of the C-flows in $\mathbb{R}^d$ (i.e., we do not perform dimensionality reduction in the second-step clustering). This second step generates a set of $\gamma_2$ smaller and more precise clusters $\{\mathcal{C}''_i\}_{i=1..\gamma_2}$.

We implement the first- and second-step clustering using the $X$-means clustering algorithm [31]. $X$-means is an efficient algorithm based on $K$-means [25], a very popular clustering algorithm. Different from $K$-means, the $X$-means algorithm does not require the user to choose the number $K$ of final clusters in advance. $X$-means runs multiple rounds of $K$-means internally and performs efficient clustering validation using the Bayesian Information Criterion [31] in order to compute the best value of $K$. $X$-means is fast and scales well with respect to the size of the dataset [31].

For the first-step (coarse-grained) clustering, we first reduce the dimensionality of the feature space from $d = 52$ features (see Section 2.5.1) into $d' = 8$ features by simply computing the mean and variance of the distribution of *fph*, *ppf*, *bpp*, and *bps* for each C-flow. Then we apply the $X$-means clustering algorithm on the obtained representation of C-flows to find the coarse-grained clusters $\{\mathcal{C}'_i\}_{i=1..\gamma_1}$. Since the size of the clusters $\{\mathcal{C}'_i\}_{i=1..\gamma_1}$ generated by the first-step clustering is relatively small, we can now afford to perform a more expensive analysis on each $\mathcal{C}'_i$. Thus, for the second-step clustering, we use all the $d = 52$ available features to represent the C-flows, and we apply the $X$-means clustering algorithm to refine the results of the first-step clustering.

Of course, since unsupervised learning is a notoriously difficult task, the results of this two-step clustering algorithm may still be not perfect. As a consequence, the C-flows related to a botnet may be grouped into some distinct clusters, which basically represent *sub-botnets*. Furthermore, a cluster that contains mostly botnet or benign C-flows may also contain some "noisy" benign or botnet C-flows, respectively. However, we would like to stress the fact that these problems are not necessarily critical and can be alleviated by performing correlation with the results of the activity-plane (A-plane) clustering (see Section 2.7).

Finally, we need to note that it is possible to bootstrap the clustering from A-plane logs. For example, one may apply clustering to only those hosts that appear in the A-plane logs (i.e., the suspicious activity logs). This may greatly reduce the workload of the C-plane clustering module, if speed is the main concern. Similarly, one may bootstrap the A-plane correlation from C-plane logs, e.g., by monitoring only clients that previously formed communication clusters, or by giving monitoring preference to those clients that demonstrate some persistent C-flow communications (assuming botnets are used for long-term purpose).

## 2.6 A-plane Clustering

In this stage, we perform two-layer clustering on activity logs. Figure 7 shows the clustering process in A-plane. For the whole list of clients that perform at least one malicious activity during one day, we first cluster them according to the types of their activities (e.g., scan, spam, and binary downloading). This is the first layer clustering. Then, for each activity type,

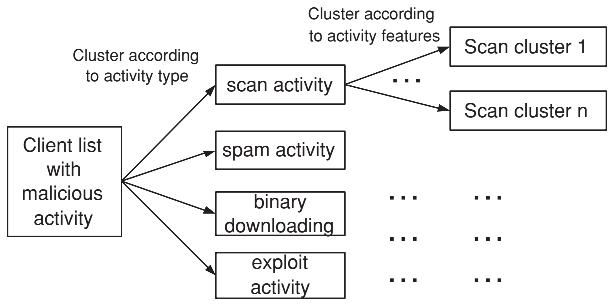

Cluster according to activity features
Cluster according to activity type
Scan cluster 1
scan activity
...
Scan cluster n
Client list with malicious activity
spam activity
binary downloading
exploit activity
...

Figure 7: A-plane clustering.

we further cluster clients according to specific activity features (the second layer clustering). For scan activity, features could include scanning ports, that is, two clients could be clustered together if they are scanning the same ports. Another candidate feature could be the target subnet/distribution, e.g., whether the clients are scanning the same subnet. For spam activity, two clients could be clustered together if their SMTP connection destinations are highly overlapped. This might not be robust when the bots are configured to use different SMTP servers in order to evade detection. One can further consider the spam content if the whole SMTP traffic is captured. To cluster spam content, one may consider the similarity of embedded URLs that are very likely to be similar with the same botnet [43], SMTP connection frequency, content entropy, and the normalized compression distance (NCD [5, 41]) on the entire email bodies. For outbound exploit activity, one can cluster two clients if they send the same type of exploit, indicated by the Snort alert SID. For binary downloading activity, two clients could be clustered together if they download similar binaries (because they download from the same URL as indicated in the command from the botmaster). A distance function between two binaries can be any string distance such as DICE used in [20] [8].

In our current implementation, we cluster scanning activities according to the destination scanning ports. For spam activity clustering, because there are very few hosts that show spamming activities in our monitored network, we simply cluster hosts together if they perform spamming (i.e., using only the first layer clustering here). For binary downloading, we configure our binary downloading monitor to capture only the first portion (packet) of the binary for efficiency reasons (if necessary, we can also capture the entire binary). We simply compare

[8] In an extreme case that bots update their binaries from different URLs (and the binaries are packed to be polymorphic thus different from each other), one should unpack the binary using tools such as Polyunpack [37] before calculating the distance. One may also directly apply normalized compression distance (NCD [5, 41]) on the original (maybe packed) binaries.

whether these early portions of the binaries are the same or not. In other words, currently, our A-plane clustering implementation utilizes relatively weak cluster features. In the future, we plan to implement clustering on more complex feature sets discussed above, which are more robust against evasion. However, even with the current weak cluster features, BotMiner already demonstrated high accuracy with a low false positive rate as shown in our later experiments.

## 2.7 Cross-plane Correlation

Once we obtain the clustering results from A-plane (activities patterns) and C-plane (communication patterns), we perform cross-plane correlation. The idea is to cross-check clusters in the two planes to find out intersections that reinforce evidence of a host being part of a botnet. In order to do this, we first compute a botnet score $s(h)$ for each host $h$ on which we have witnessed at least one kind of suspicious activity. We filter out the hosts that have a score below a certain detection threshold $\theta$, and then group the remaining most suspicious hosts according to a similarity metric that takes into account the A-plane and C-plane clusters these hosts have in common.

We now explain how the botnet score is computed for each host. Let $H$ be the set of hosts reported in the output of the A-plane clustering module, and $h \in H$. Also, let $\mathcal{A}^{(h)} = \{A_i\}_{i=1..m_h}$ be the set of $m_h$ A-clusters that contain $h$, and $\mathcal{C}^{(h)} = \{C_i\}_{i=1..n_h}$ be the set of $n_h$ C-clusters that contain $h$. We compute the botnet score for $h$ as

$$s(h) = \sum_{\substack{i,j \\ j>i \\ t(A_i)\neq t(A_j)}} w(A_i)w(A_j)\frac{|A_i \cap A_j|}{|A_i \cup A_j|} + \sum_{i,k} w(A_i)\frac{|A_i \cap C_k|}{|A_i \cup C_k|}, \quad (1)$$

where $A_i, A_j \in \mathcal{A}^{(h)}$ and $C_k \in \mathcal{C}^{(h)}$, $t(A_i)$ is the type of activity cluster $A_i$ refers to (e.g., scanning or spamming), and $w(A_i) \geqslant 1$ is an *activity weight* assigned to $A_i$. $w(A_i)$ assigns higher values to "strong" activities (e.g., spam and exploit) and lower values to "weak" activities (e.g., scanning and binary download).

$h$ will receive a high score if it has performed multiple types of suspicious activities, and if other hosts that were clustered with $h$ also show the same multiple types of activities. For example, assume that $h$ performed scanning and then attempted to exploit a machine outside the monitored network. Let $A_1$ be the cluster of hosts that were found to perform scanning and were grouped with $h$ in the same cluster. Also, let $A_2$ be a cluster related to exploit activities that includes $h$ and other hosts that performed similar activities. A larger overlap between $A_1$ and $A_2$ would result in a higher score being assigned to $h$. Similarly, if $h$ belongs to A-clusters that have a large overlap with C-clusters, then it means that the hosts clustered together with $h$ share similar activities as well as similar communication patterns.

Given a predefined detection threshold $\theta$, we consider all the hosts $h \in H$ with $s(h) > \theta$ as (likely) bots, and filter out the hosts whose scores do not exceed $\theta$. Now, let $B \subseteq H$ be the set of detected bots, $\mathcal{A}^{(B)} = \{A_i\}_{i=1..m_B}$ be the set of A-clusters that each contains at least one bot $h \in B$, and $\mathcal{C}^{(B)} = \{C_i\}_{i=1..n_B}$ be the set of C-clusters that each contains at least one bot $h \in B$. Also, let $\mathcal{K}^{(B)} = \mathcal{A}^{(B)} \cup \mathcal{C}^{(B)} = \{K_i^{(B)}\}_{i=1..(m_B+n_B)}$ be an ordered union/set of A- and C-clusters. We then describe each bot $h \in B$ as a binary vector $b(h) \in \{0,1\}^{|\mathcal{K}^{(B)}|}$, whereby the $i$-th element $b_i = 1$ if $h \in K_i^{(B)}$, and $b_i = 0$ otherwise. Given this representation, we can define the following similarity between bots $h_i$ and $h_j$ as

$$sim(h_i, h_j) = \sum_{k=1}^{m_B} I(b_k^{(i)} = b_k^{(j)}) + I(\sum_{k=m_B+1}^{m_B+n_B} I(b_k^{(i)} = b_k^{(j)}) \geq 1), \quad (2)$$

where we use $b^{(i)} = b(h_i)$ and $b^{(j)} = b(h_j)$, for the sake of brevity. $I(X)$ is the indication function, which equals to one when the boolean argument $X$ is true, and equals to zero when $X$ is false. The intuition behind this metric is that if two hosts appear in the same activity clusters and in at least one common C-cluster, they should be clustered together.

This definition of similarity between hosts gives us the opportunity to apply hierarchical clustering. This allows us to build a dendrogram, i.e., a tree like graph (see Figure 8) that encodes the relationships among the bots. We use the Davies-Bouldin (DB) validation index [21] to find the best dendrogram cut, which produces the most compact and well separated clusters. The obtained clusters group bots in (sub-) botnets. Figure 8 shows a (hypothetical) example. Assuming that the best cut suggested by the DB index is the one at height 90, we would obtain two botnets, namely $\{h_8, h_3, h_5\}$, and $\{h_4, h_6, h_9, h_2, h_1, h_7\}$.

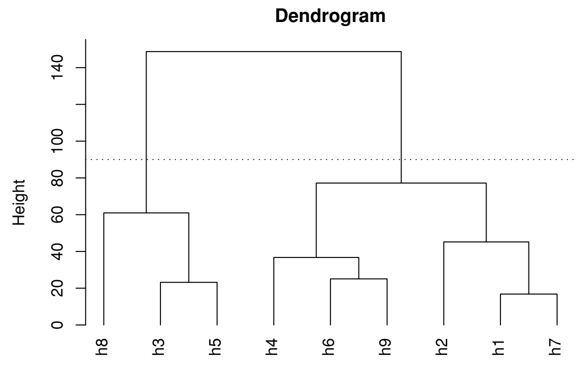


Figure 8: Example of hierarchical clustering for botnet detection.

In our current implementation, we simply set weight $w(A_i) = 1$ for all $i$ and $\theta = 0$, which essentially means that we will consider all hosts that appear in two different types of A-clusters and/or in both A- and C-clusters as suspicious candidates for further hierarchical clustering.

## 3 Experiments

To evaluate our BotMiner detection framework and prototype system, we have tested its performance on several real-world network traffic traces, including both (presumably) normal data from our campus network and collected botnet data.

### 3.1 Experiment Setup and Data Collection

We set up traffic monitors to work on a span port mirroring a backbone router at the campus network of the College of Computing at Georgia Tech. The traffic rate is typically 200Mbps-300Mbps at daytime. We ran the C-plane and A-plane monitors for a continuous 10-day period in late 2007. A random sampling of the network trace shows that the traffic is very diverse, containing many normal application protocols, such as `HTTP`, `SMTP`, `POP`, `FTP`, `SSH`, `NetBios`, `DNS`, `SNMP`, IM (e.g., `ICQ`, `AIM`), P2P (e.g., `Gnutella`, `Edonkey`, `bittorrent`), and `IRC`. This serves as a good background to test the false positives and detection performance on a normal network with rich application protocols.

We have collected a total of eight different botnets covering `IRC, HTTP` and P2P. Table 1 lists the basic information about these traces.

We re-used two `IRC` and two `HTTP` botnet traces introduced in [20], i.e., `V-Spybot`, `V-Sdbot`, `B-HTTP-I`, and `B-HTTP-II`. In short, `V-Spybot` and `V-Sdbot` are generated by executing modified bot code (Spybot and Sdbot [6]) in a fully controlled virtual network. They contain four Windows XP/2K IRC bot clients, and last several minutes. `B-HTTP-I` and `B-HTTP-II` are generated based on the description of Web-based C&C communications in [24, 39]. Four bot clients communicate with a controlled server and execute the received command (e.g., `spam`). In `B-HTTP-I`, the bot contacts the server periodically (about every five minutes) and the whole trace lasts for about 3.6 hours. In `B-HTTP-II`, we have a more stealthy C&C communication where the bot waits a random time between zero to ten minutes each time before it visits the server, and the whole trace lasts for 19 hours. These four traces are renamed as `Botnet-IRC-spybot`, `Botnet-IRC-sdbot`, `Botnet-HTTP-1`, and `Botnet-HTTP-2`, respectively. In addition, we also generated a new `IRC` botnet trace that lasts for a longer time (a whole day) using modified Rbot [3] source code. Again this is generated in a controlled virtual network with four Windows clients and one `IRC` server. This trace is labeled as `Botnet-IRC-rbot`.

We also obtained a real-world `IRC`-based botnet C&C trace that was captured in the wild in 2004, labeled as `Botnet-IRC-N`. The trace contains about 7-minute `IRC` C&C communications, and has hundreds of bots connected to the `IRC` C&C server. The botmaster set the command "`.scan.startall`" in the TOPIC of the channel. Thus, every bot would begin to propagate through scanning once joining the channel. They report their successful transfer of binary to some machines, and also report the machines that have been exploited. We believe this could be a variant of Phatbot [6]. Although we obtained only the `IRC` C&C traffic, we hypothesize that the scanning activities are easy to detect given the fact that bots are performing scanning commands in order to propagate. Thus, we assume we have an A-plane cluster with the botnet members because we want to see if we can still capture C-plane clusters and obtain cross-plane correlation results.

Finally, we obtained a real-world trace containing two P2P botnets, Nugache [28] and Storm [18, 23]. The trace lasts for a whole day, and there are 82 Nugache bots and 13 Storm bots in the trace. It was captured from a group of honeypots running in the wild in late 2007. Each instance is running in Wine (an open source implementation of the Windows API on top of Unix/Linux) instead of a virtual or physical machine. Such a set-up is known as winobot [12] and is used by researchers to track botnets. By using a lightweight emulation environment (Wine), winobots can run hundreds and thousands of black-box instances of a given malware. This allows one to participate in a P2P botnet *en mass*. Nugache is a TCP-based P2P bot that performs encrypted communications on port 8. Storm, originating in January of 2007, is one of the very few known UDP based P2P bots. It is based on the Kademlia [30] protocol and makes use of the Overnet network [2] to locate related data (e.g., commands). Storm is well-known as a spam botnet with a huge number of infected hosts [27]. In the implementation of winobot, several malicious capabilities such as sending spam are disabled for legality reason, thus we can not observe spam traffic from the trace. However, we ran a full version of Storm on a VM-based honeypot (instead of Wine environment) and easily observed that it kept sending a huge amount of spam traffic, which makes the A-plane monitoring quite easy. Similarly, when running Nugache on a VM-based honeypot, we observed scanning activity to port 8 because it attempted to connect to its seeding peers but failed a lot of times (because the peers may not be available). Thus, we can detect and cluster A-plane activities for these P2P botnets.

| Trace | Size | Duration | Pkt | TCP/UDP flows | Botnet clients | C&C server |
|---|---|---|---|---|---|---|
| Botnet-IRC-rbot | 169MB | 24h | 1,175,083 | 180,988 | 4 | 1 |
| Botnet-IRC-sdbot | 66KB | 9m | 474 | 19 | 4 | 1 |
| Botnet-IRC-spybot | 15MB | 32m | 180,822 | 147,945 | 4 | 1 |
| Botnet-IRC-N | 6.4MB | 7m | 65,111 | 5635 | 259 | 1 |
| Botnet-HTTP-1 | 6MB | 3.6h | 65,695 | 2,647 | 4 | 1 |
| Botnet-HTTP-2 | 37MB | 19h | 395,990 | 9,716 | 4 | 1 |
| Botnet-P2P-Storm | 1.2G | 24h | 59,322,490 | 5,495,223 | 13 | P2P |
| Botnet-P2P-Nugache | 1.2G | 24h | 59,322,490 | 5,495,223 | 82 | P2P |

Table 1: Collected botnet traces, covering `IRC`, `HTTP` and P2P based botnets. Storm and Nugache share the same file, so the statistics of the whole file are reported.

| Trace | Pkts | Flows | Filtered by `F1` | Filtered by `F2` | Filtered by `F3` | Flows after filtering | C-flows (TCP/UDP) |
|---|---|---|---|---|---|---|---|
| Day-1 | 5,178,375,514 | 23,407,743 | 20,727,588 | 939,723 | 40,257 | 1,700,175 | 66,981 / 132,333 |
| Day-2 | 7,131,674,165 | 29,632,407 | 27,861,853 | 533,666 | 25,758 | 1,211,130 | 34,691 / 96,261 |
| Day-3 | 9,701,255,613 | 30,192,645 | 28,491,442 | 513,164 | 24,329 | 1,163,710 | 39,744 / 94,081 |
| Day-4 | 14,713,667,172 | 35,590,583 | 33,434,985 | 600,901 | 33,958 | 1,520,739 | 73,021 / 167,146 |
| Day-5 | 11,177,174,133 | 56,235,380 | 52,795,168 | 1,323,475 | 40,016 | 2,076,721 | 57,664 / 167,175 |
| Day-6 | 9,950,803,423 | 75,037,684 | 71,397,138 | 1,464,571 | 51,931 | 2,124,044 | 59,383 / 176,210 |
| Day-7 | 10,039,871,506 | 109,549,192 | 105,530,316 | 1,614,158 | 56,688 | 2,348,030 | 55,023 / 150,211 |
| Day-8 | 11,174,937,812 | 96,364,123 | 92,413,010 | 1,578,215 | 60,768 | 2,312,130 | 56,246 / 179,838 |
| Day-9 | 9,504,436,063 | 62,550,060 | 56,516,281 | 3,163,645 | 30,581 | 2,839,553 | 25,557 / 164,986 |
| Day-10 | 11,071,701,564 | 83,433,368 | 77,601,188 | 2,964,948 | 27,837 | 2,839,395 | 25,436 / 154,294 |

Table 2: C-plane traffic statistics, basic results of filtering, and C-flows.

## 3.2 Evaluation Results

Table 2 lists the statistics for the 10 days of network data we used to validate our detection system. For each day there are around 5-10 billion packets (TCP and UDP) and 30-100 million flows. Table 2 shows the results of several steps of filtering. The first step of filtering (filter rule `F1`) seems to be the most effective filter in terms of data volume reduction. `F1` filters out those flows that are not initiated from internal hosts to external hosts, and achieves about 90% data volume reduction. The is because most of the flows are within the campus network (i.e., they are initiated from internal hosts towards other internal hosts). `F2` further filters out around 0.5-3 million of non-completely-established flows. `F3` further reduces the data volume by filtering out another 30,000 flows. After applying all the three steps of filtering, there are around 1 to 3 million flows left per day. We converted these remaining flows into C-flows as described in Section 2.5, and obtained around 40,000 TCP C-flows and 130,000 UDP C-flows per day.

We then performed two-step clustering on C-flows as described in Section 2.5. Table 3 shows the clustering results and false positives (number of clusters that are not botnets). The results for the first 5 days are related to both TCP and UDP traffic, whereas in the last 5 days we focused on only TCP traffic.

It is easy to see from Table 3 that there are thousands of C-clusters generated each day. In addition, there are several thousand activity logs generated from A-plane monitors. Since we use relatively weak monitor modules, it is not surprising that we have this many activity logs. Many logs report binary downloading events or scanning activities. We cluster these activity logs according to their activity features. As explained early, we are interested in groups of machines that perform activities in a similar/coordinated way. Therefore, we filter out the A-clusters that contain only one host. This simple filtering rule allows us to obtain a small number of A-clusters and reduce the overall false positive rate of our botnet detection system.

Afterwards, we apply cross-plane correlation. We assume that the traffic we collected from our campus network is normal. In order to verify this assumption we used state-of-the-art botnet detection techniques like BotHunter [19] and BotSniffer [20]. Therefore, any cluster generated as a result of the cross-plane correlation is considered as a *false positive cluster*. It is easy to see from Table 3 that there are very few such false positive clusters every day (from zero to four). Most of these clusters contain only two clients (i.e., they induce two false positives). In three out of ten days no false positive was reported. In both Day-2 and Day-3, the cross-correlation produced one false positive cluster containing two hosts. Two false positive clusters were reported in each day from Day-5 to Day-8. In Day-4, the cross-plane correlation produced four false positive clusters.

For each day of traffic, the last column of Table 3 shows the false positive rate (FP rate), which is calculated as the fraction of IP addresses reported in the false positive clusters over the total number of distinct normal clients appearing in that day. After further analysis we found that many of these false positives are caused by clients performing binary downloading from websites not present in our whitelist. In practice, the number of false positives may be reduced by implementing a better binary downloading monitor and clustering mod-

| Trace | Step-1 C-clusters | Step-2 C-clusters | A-plane logs | A-clusters | False Positive Clusters | FP Rate |
|---|---|---|---|---|---|---|
| Day-1 (TCP/UDP) | 1,374 | 4,958 | 1,671 | 1 | 0 | 0 (0/878) |
| Day-2 (TCP/UDP) | 904 | 2,897 | 5,434 | 1 | 1 | 0.003 (2/638) |
| Day-3 (TCP/UDP) | 1,128 | 2,480 | 4,324 | 1 | 1 | 0.003 (2/692) |
| Day-4 (TCP/UDP) | 1,528 | 4,089 | 5,483 | 4 | 4 | 0.01 (9/871) |
| Day-5 (TCP/UDP) | 1,051 | 3,377 | 6,461 | 5 | 2 | 0.0048 (4/838) |
| Day-6 (TCP) | 1,163 | 3,469 | 6,960 | 3 | 2 | 0.008 (7/877) |
| Day-7 (TCP) | 954 | 3,257 | 6,452 | 5 | 2 | 0.006 (5/835) |
| Day-8 (TCP) | 1,170 | 3,226 | 8,270 | 4 | 2 | 0.0091 (8/877) |
| Day-9 (TCP) | 742 | 1,763 | 7,687 | 2 | 0 | 0 (0/714) |
| Day-10 (TCP) | 712 | 1,673 | 7,524 | 0 | 0 | 0 (0/689) |

Table 3: C-plane and A-plane clustering results.

| Botnet | Number of Bots | Detected? | Clustered Bots | Detection Rate | False Positive Clusters/Hosts | FP Rate |
|---|---|---|---|---|---|---|
| IRC-rbot | 4 | YES | 4 | 100% | 1/2 | 0.003 |
| IRC-sdbot | 4 | YES | 4 | 100% | 1/2 | 0.003 |
| IRC-spybot | 4 | YES | 3 | 75% | 1/2 | 0.003 |
| IRC-N | 259 | YES | 258 | 99.6% | 0 | 0 |
| HTTP-1 | 4 | YES | 4 | 100% | 1/2 | 0.003 |
| HTTP-2 | 4 | YES | 4 | 100% | 1/2 | 0.003 |
| P2P-Storm | 13 | YES | 13 | 100% | 0 | 0 |
| P2P-Nugache | 82 | YES | 82 | 100% | 0 | 0 |

Table 4: Botnet detection results using BotMiner.

ule, e.g., by capturing the entire binary and performing content inspection (using either anomaly-based detection systems [38] or signature-based AV tools).

In order to validate the detection accuracy of BotMiner, we overlaid botnet traffic to normal traffic. We consider one botnet trace at a time and overlay it to the entire normal traffic trace of Day-2. We simulate a near-realistic scenario by constructing the test dataset as follows. Let $n$ be the number of distinct bots in the botnet trace we want to overlay to normal traffic. We randomly select $n$ distinct IP addresses from the normal traffic trace and map them to the $n$ IP addresses of the bots. That is, we replace an $IP_i$ of a normal machine with the $IP_i$ of a bot. In this way, we obtain a dataset of mixed normal and botnet traffic where a set of $n$ machines show both normal and botnet-related behavior. Table 4 reports the detection results for each botnet.

Table 4 shows that BotMiner is able to detect all eight botnets. We verified whether the members in the reported clusters are actually bots or not. For 6 out of 8 botnets, we obtained 100% detection rate, i.e., we successfully identified all the bots within the 6 botnets. For example, in the case of P2P botnets (`Botnet-P2P-Nugache` and `Botnet-P2P-Storm`), BotMiner correctly generated a cluster containing all the botnet members. In the case of `Botnet-IRC-spybot`, BotMiner correctly detected a cluster of bots. However, one of the bots belonging to the botnet was not reported in the cluster, which means that the detector generated a false negative. `Botnet-IRC-N` contains 259 bot clients. BotMiner was able to identify 258 of the bots in one cluster, whereas one of the bots was not detected. Therefore, in this case BotMiner had a detection rate of 99.6%.

There were some cases in which BotMiner also generated a false positive cluster containing two normal hosts. We verified that these two normal hosts in particular were also responsible for the false positives generated during the analysis of the Day-2 normal traffic (see Table 3).

As we can see, BotMiner performs quite well in our experiments, showing a very high detection rate with relatively few false positives in real-world network traces.

## 4 Limitations and Potential Solutions

Like any intrusion/anomaly detection system, BotMiner is not perfect or complete. It is likely that once adversaries know our detection framework and implementation, they might find some ways to evade detection, e.g., by evading the C-plane and A-plane monitoring and clustering, or the cross-plane correlation analysis. We now address these limitations and discuss possible solutions.

### 4.1 Evading C-plane Monitoring and Clustering

Botnets may try to utilize a legitimate website (e.g., `Google`) for their C&C purpose in attempt to evade detection. Evasion would be successful in this case if we whitelisted such legitimate websites to reduce the volume of monitored traffic and improve the efficiency of our detection system. However, if a legitimate website, say `Google`, is used as a means to locate a secondary URL for actual command hosting or binary downloading, botnets may not be able to hide this secondary URL and the corresponding communications. Therefore, clustering of network traffic towards the server pointed by this secondary URL will likely allow us to detect the bots. Also, whitelisting is just an optional operation. One may easily choose not to use whitelisting to avoid such kind of evasion attempts (of course, in this case one may face the trade-off between accuracy and efficiency).

Botnet members may attempt to intentionally manipulate their communication patterns to evade our C-plane clustering. The easiest thing is to switch to multiple C&C servers. However, this does not help much to evade our detection because such peer communications could still be clustered together just like how we cluster P2P communications. A more advanced way is to randomize each individual communication pattern, for example by randomizing the number of packets per flow (e.g., by injecting random packets in a flow), and the number of bytes per packet (e.g., by padding random bytes in a packet). However, such randomization may introduce similarities among botnet members if we measure the distribution and entropy of communication features. Also, this randomization may raise suspicion because normal user communications may not have such randomized patterns. Advanced evasion may be attempted by bots that try to mimic the communication patterns of normal hosts, in a way similar to polymorphic blending attacks [15]. Furthermore, bots could use covert channes [1] to hide their actual C&C communications. We acknowledge that, generally speaking, communication randomization, mimicry attacks and covert channel represent limitations for all traffic-based detection approaches, including BotMiner's C-plane clustering technique. By incorporating more detection features such as content inspection and host level analysis, the detection system may make evasion more difficult.

Finally, we note that if botnets are used to perform multiple tasks (in A-plane), we may still detect them even when they can evade C-plane monitoring and analysis. By using the scoring algorithm described in Section 2.7, we can perform cross clustering analysis among multiple activity clusters (in A-plane) to accumulate the suspicious score needed to claim the existence of botnets. Thus, we may even *not* require C-plane analysis if there is already a *strong* cross-cluster correlation among different types of malicious activities in A-plane. For example, if the same set of hosts involve several types of A-plane clusters (e.g., they send spams, scan others, and/or download the same binaries), they can be reported as botnets because those behaviors, by themselves, are highly suspicious and most likely indicating botnets behaviors [19,20].

### 4.2 Evading A-plane Monitoring and Clustering

Malicious activities of botnets are unlikely or relatively hard to change as long as the botmaster wants the botnets to perform "useful" tasks. However, the botmaster can attempt to evade BotMiner's A-plane monitoring and clustering in several ways.

Botnets may perform very stealthy malicious activities in order to evade the detection of A-plane monitors. For example, they can scan very slowly (e.g., send one scan per hour), send spam very slowly (e.g., send one spam per day). This will evade our monitor sensors. However, this also puts a limit on the utility of bots.

In addition, as discussed above, if the botmaster commands each bot *randomly and individually* to perform different task, the bots are not different from previous generations of isolated, individual malware instances. This is unlikely the way a bot*net* is used in practice. A more advanced evasion is to differentiate the bots and avoid commanding bots in the same monitored network the same way. This will cause additional effort and inconvenience for the botmaster. To defeat such an evasion, we can deploy distributed monitors on the Internet to cover a larger monitored space.

Note, if the botmaster takes the extreme action of randomizing/individualizing both the C&C communications and attack activities of each bots, then these bots are probably not part of a bot*net* according to our specific definition because the bots are not performing similar/coordinated commanded activities. Orthogonal to the *horizontal correlation* approaches such as BotMiner to detect a bot*net*, we can always use complementary systems like BotHunter [19] that examine the behavior history of *distinct* host for a dialog or *vertical correlation* based approach to detect *individual* bots.

### 4.3 Evading Cross-plane Analysis

A botmaster can command the bots to perform an extremely delayed task (e.g., delayed for days after receiving commands). Thus, the malicious activities and C&C communications are in different days. If only using one day's data, we may not be able to yield cross-plane clusters. As a solution, we may use multiple-day data and cross check back several days. Although this has the hope of capturing these botnets, it may also suffer from generating more false positives. Clearly, there is a trade-off. The botmaster also faces the trade-off because a very slow C&C essentially impedes the efficiency in controlling/coordinating the bot army. Also, a bot infected machine may be disconnected from the Internet or be powered off by the users during the delay and become unavailable to the botmaster.

In summary, while it is possible that a botmaster can find a way to exploit the limitations of BotMiner, the convenience or the efficiency of botnet C&C and the utility of the botnet also suffer. Thus, we believe that our protocol- and structure-independent detection framework represents a significant advance in botnet detection.

## 5 Related Work

To collect and analyze bots, researchers widely utilize honeypot techniques [4,16,32]. Freiling et al. [16] used honeypots to study the problem of botnets. Nepenthes [4] is a special honeypot tool for automatic malware sample collection. Rajab et al. [32] provided an in-depth measurement study of the current botnet activities by conducting a multi-faceted approach to collect bots and track botnets. Cooke et al. [10] conducted several basic studies of botnet dynamics. In [13], Dagon et al. proposed to use DNS sinkholing technique for botnet study and pointed out the global diurnal behavior of botnets. Barford and Yegneswaran [6] provided a detailed study on the code base of several common bot families. Collins et al. [9] presented their observation of a relationship between botnets and scanning/spamming activities.

Several recent papers proposed different approaches to detect botnets. Ramachandran et al. [34] proposed using DNSBL counter-intelligence to find botnet members that generate spams. This approach is useful for just certain types of spam botnets. In [35], Reiter and Yen proposed a system TAMD to detect malware (including botnets) by aggregating traffic that shares the same external destination, similar payload, and that involves internal hosts with similar OS platforms. TAMD's aggregation method based on destination networks focuses on networks that experience an increase in traffic as compared to a historical baseline. Different from BotMiner that focuses on botnet detection, TAMD aims to detect a broader range of malware. Since TAMD's aggregation features are different from BotMiner's (in which we cluster similar communication patterns and similar malicious activity patterns), TAMD and BotMiner can complement each other in botnet and malware detection. Livadas et al. [29,40] proposed a machine learning based approach for botnet detection using some general network-level traffic features of chat-like protocols such as `IRC`. Karasaridis et al. [26] studied network flow level detection of IRC botnet controllers for backbone networks. The above two are similar to our work in C-plane clustering but different in many ways. First, they are used to detect IRC-based botnet (by matching a known IRC traffic profile), while we do not have the assumption of known C&C protocol profiles. Second, we use a different feature set on a new communication flow (C-flow) data format instead of traditional network flow. Third, we consider both C-plane and A-plane information instead of just flow records.

Rishi [17] is a signature-based `IRC` botnet detection system by matching known IRC bot nickname patterns. Binkley and Singh [7] proposed combining IRC statistics and TCP work weight for the detection of IRC-based botnets. In [19], we described BotHunter, which is a passive bot detection system that uses dialog correlation to associate IDS events to a user-defined bot infection dialog model. Different from BotHunter's *dialog correlation* or *vertical correlation* that mainly examines the behavior history associated with each *distinct* host, BotMiner utilizes a *horizontal correlation* approach that examines correlation *across* multiple hosts. BotSniffer [20] is an anomaly-based botnet C&C detection system that also utilizes *horizontal correlation*. However, it is used mainly for detecting *centralized* C&C activities (e.g., `IRC` and `HTTP`).

The aforementioned systems are mostly limited to specific botnet protocols and structures, and many of them work only on IRC-based botnets. BotMiner is a novel general detection system that does not have such limitations and can greatly complement existing detection approaches.

## 6 Conclusion & Future Work

Botnet detection is a challenging problem. In this paper, we proposed a novel network anomaly-based botnet detection system that is independent of the protocol and structure used by botnets. Our system exploits the essential definition and properties of botnets, i.e., bots within the same botnet will exhibit similar C&C communication patterns and similar malicious activities patterns. In our experimental evaluation on many real-world network traces, BotMiner shows excellent detection accuracy on various types of botnets (including `IRC`-based, `HTTP`-based, and P2P-based botnets) with a very low false positive rate on normal traffic.

It is likely that future botnets (especially P2P botnets) may utilize evasion techniques to avoid detection, as discussed in Section 4. In our future work, we will study new techniques to monitor/cluster communication and activity patterns of botnets, and these techniques are intended to be more robust to evasion attempts. In addition, we plan to further improve the efficiency of the C-flow converting and clustering algorithms, combine different correlation techniques (e.g., vertical correlation and horizontal correlation), and develop new real-time detection systems based on a layered design using sampling techniques to work in very high speed and very large network environments.

## Acknowledgments

We would like to thank David Dagon and Yan Chen for their help in providing some of the evaluation data in our experiments. We thank Robert Edmonds for his support on using `fcapture`. In addition, we thank Angelos Stavrou, Rachna Dhamija, and anonymous reviewers for their insightful comments and feedback. This material is based upon work supported in part by the National Science Foundation under Grants CCR-0133629, CNS-0627477, and CNS-0716570, by the U.S. Army Research Office under Grant W911NF0610042, and by the Air Force Research Laboratory (AFRL) under Grant FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the U.S. Army Research Office, and the Air Force Research Laboratory.

## References

[1] A guide to understanding covert channel analysis of trusted systems, version 1. NCSC-TG-030, Library No. S-240,572, National Computer Security Center, November 1993.

[2] Overnet. `http://en.wikipedia.org/wiki/Overnet`, 2008.

[3] P. Bacher, T. Holz, M. Kotter, and G. Wicherski. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`, 2005.

[4] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, September 2006.

[5] M. Bailey, J. Oberheide, J. Andersen, M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, 2007.

[6] P. Barford and V. Yegneswaran. An Inside Look at Botnets. Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.

[7] J. R. Binkley and S. Singh. An algorithm for anomaly-based botnet detection. In *Proceedings of USENIX SRUTI'06*, pages 43–48, July 2006.

[8] K. Chiang and L. Lloyd. A case study of the rustock rootkit and spam bot. In *Proceedings of USENIX HotBots'07*, 2007.

[9] M. Collins, T. Shimeall, S. Faber, J. Janies, R. Weaver, M. D. Shon, and J. Kadane. Using uncleanliness to predict future botnet addresses,. In *Proceedings of ACM/USENIX Internet Measurement Conference (IMC'07)*, 2007.

[10] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of USENIX SRUTI'05*, 2005.

[11] Cyber-TA. BotHunter Free Internet Distribution Page. `http://www.cyber-ta.org/BotHunter`, 2008.

[12] D. Dagon, G. Gu, C. Lee, and W. Lee. A taxonomy of botnet structures. In *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[13] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using timezones. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)*, January 2006.

[14] N. Daswani and M. Stoppelman. The anatomy of clickbot.a. In *Proceedings of USENIX HotBots'07*, 2007.

[15] P. Fogla, M. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee. Polymorphic blending attack. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, 2006.

[16] F. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-cause Methodology to Prevent Denial of Service Attacks. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS'05)*, 2005.

[17] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *Proceedings of USENIX HotBots'07*, 2007.

[18] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of USENIX HotBots'07*, 2007.

[19] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, 2007.

[20] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[21] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2-3):107–145, 2001.

[22] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Detection and mitigation of fast-flux service networks. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[23] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'08)*, 2008.

[24] N. Ianelli and A. Hackworth. Botnets as a vehicle for online crime. http://www.cert.org/archive/pdf/Botnets.pdf, 2005.

[25] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computer Survey*, 31(3):264–323, 1999.

[26] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *Proceedings of USENIX HotBots'07*, 2007.

[27] B. Krebs. Storm worm dwarfs world's top supercomputers. `http://blog.washingtonpost.com/securityfix/2007/08/storm_worm_dwarfs_worlds_top_s_1.html`, 2007.

[28] R. Lemos. Bot software looks to improve peerage. `Http://www.securityfocus.com/news/11390`, 2006.

[29] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer. Using machine learning techniques to identify botnet traffic. In *Proceedings of the 2nd IEEE LCN Workshop on Network Security (WoNS'2006)*, 2006.

[30] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[31] D. Pelleg and A. W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[32] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multi-faceted approach to understanding the botnet phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference (IMC'06)*, Brazil, October 2006.

[33] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM'06*, 2006.

[34] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using DNSBL counter-intelligence. In *Proceedings of USENIX SRUTI'06*, 2006.

[35] M. K. Reiter and T.-F. Yen. Traffic aggregation for malware detection. In *Proceedings of the Fifth GI International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'08)*, 2008.

[36] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of USENIX LISA'99*, 1999.

[37] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[38] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[39] SecureWorks. Bobax trojan analysis. http://www.secureworks.com/research/threats/bobax/, 2004.

[40] W. T. Strayer, R. Walsh, C. Livadas, and D. Lapsley. Detecting botnets with tight command and control. In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN'06)*, 2006.

[41] S. Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320, 2007.

[42] T. Werner. PE Hunter. `http://honeytrap.mwcollect.org/pehunter`, 2007.

[43] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, I. Osipkov, G. Hulten, and J. Tygar. Characterizing botnets from email spam records. In *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'08)*, 2008.

[44] J. Zhuge, T. Holz, X. Han, J. Guo, and W. Zou. Characterizing the irc-based botnet phenomenon. Peking University & University of Mannheim Technical Report, 2007.

# Measurement and Classification of Humans and Bots in Internet Chat

Steven Gianvecchio, Mengjun Xie, Zhenyu Wu, and Haining Wang
*Department of Computer Science*
*The College of William and Mary*
*{srgian, mjxie, adamwu, hnw}@cs.wm.edu*

## Abstract

The abuse of chat services by automated programs, known as chat bots, poses a serious threat to Internet users. Chat bots target popular chat networks to distribute spam and malware. In this paper, we first conduct a series of measurements on a large commercial chat network. Our measurements capture a total of 14 different types of chat bots ranging from simple to advanced. Moreover, we observe that human behavior is more complex than bot behavior. Based on the measurement study, we propose a classification system to accurately distinguish chat bots from human users. The proposed classification system consists of two components: (1) an entropy-based classifier and (2) a machine-learning-based classifier. The two classifiers complement each other in chat bot detection. The entropy-based classifier is more accurate to detect unknown chat bots, whereas the machine-learning-based classifier is faster to detect known chat bots. Our experimental evaluation shows that the proposed classification system is highly effective in differentiating bots from humans.

## 1 Introduction

Internet chat is a popular application that enables real-time text-based communication. Millions of people around the world use Internet chat to exchange messages and discuss a broad range of topics on-line. Internet chat is also a unique networked application, because of its human-to-human interaction and low bandwidth consumption [9]. However, the large user base and open nature of Internet chat make it an ideal target for malicious exploitation.

The abuse of chat services by automated programs, known as *chat bots*, poses a serious threat to on-line users. Chat bots have been found on a number of chat systems, including commercial chat networks, such as AOL [15, 29], Yahoo! [19, 25, 26, 28, 34] and MSN [16], and open chat networks, such as IRC and Jabber. There are also reports of bots in some non-chat systems with chat features, including online games, such as World of Warcraft [7, 32] and Second Life [27]. Chat bots exploit these on-line systems to send spam, spread malware, and mount phishing attacks.

So far, the efforts to combat chat bots have focused on two different approaches: (1) keyword-based filtering and (2) human interactive proofs. The keyword-based message filters, used by third party chat clients [42, 43], suffer from high false negative rates because bot makers frequently update chat bots to evade published keyword lists. The use of human interactive proofs, such as CAPTCHAs [1], is also ineffective because bot operators assist chat bots in passing the tests to log into chat rooms [25, 26]. In August 2007, Yahoo! implemented CAPTCHA to block bots from entering chat rooms, but bots are still able to enter chat rooms in large numbers. There are online petitions against both AOL and Yahoo! [28, 29], requesting that the chat service providers address the growing bot problem. While on-line systems are besieged with chat bots, no systematic investigation on chat bots has been conducted. The effective detection system against chat bots is in great demand but still missing.

In the paper, we first perform a series of measurements on a large commercial chat network, Yahoo! chat, to study the behaviors of chat bots and humans in on-line chat systems. Our measurements capture a total of 14 different types of chat bots. The different types of chat bots use different triggering mechanisms and text obfuscation techniques. The former determines message timing, and the latter determines message content. Our measurements also reveal that human behavior is more complex than bot behavior, which motivates the use of entropy rate, a measure of complexity, for chat bot classification. Based on the measurement study, we propose a classification system to accurately distinguish chat bots from humans. There are two main components in our classification system: (1) an entropy classifier and (2) a machine-learning classifier. Based on the characteristics of message time and size, the entropy classifier measures the complexity of chat flows and then classifies them as bots or humans. In contrast, the machine-learning classifier is mainly based on message content for detection. The two classifiers complement each other in chat bot detection. While the entropy classifier requires more messages for detection and, thus, is slower, it is more accurate to detect unknown chat bots. Moreover, the entropy classifier helps train the machine-learning classifier. The machine learning classifier requires less messages for detection and, thus, is faster, but cannot detect most unknown bots. By combining the entropy classifier and the machine-learning classifier, the proposed classification system is highly effective to capture chat bots, in terms of accuracy and speed. We conduct experimental tests on the classification system, and the results validate its efficacy on chat bot detection.

The remainder of this paper is structured as follows. Section 2 covers background on chat bots and related work. Section 3 details our measurements of chat bots and humans. Section 4 describes our chat bot classification system. Section 5 evaluates the effectiveness of our approach for chat bot detection. Finally, Section 6 concludes the paper and discusses directions for our future work.

## 2 Background and Related Work

### 2.1 Chat Systems

Internet chat is a real-time communication tool that allows on-line users to communicate via text in virtual spaces, called chat rooms or channels. There are a number of protocols that support chat [17], including IRC, Jabber/XMPP, MSN/WLM (Microsoft), OSCAR (AOL), and YCHT/YMSG (Yahoo!). The users connect to a chat server via chat clients that support a certain chat protocol, and they may browse and join many chat rooms featuring a variety of topics. The chat server relays chat messages to and from on-line users. A chat service with a large user base might employ multiple chat servers. In addition, there are several multi-protocol chat clients, such as Pidgin (formerly GAIM) and Trillian, that allow a user to join different chat systems.

Although IRC has existed for a long time, it has not gained mainstream popularity. This is mainly because its console-like interface and command-line-based operation are not user-friendly. The recent chat systems improve user experience by using graphic-based interfaces, as well as adding attractive features such as avatars, emoticons, and audio-video communication capabilities. Our study is carried out on the Yahoo! chat network, one of the largest and most popular commercial chat systems.

Yahoo! chat uses proprietary protocols, in which the chat messages are transmitted in plain-text, while commands, status and other meta data are transmitted as encoded binary data. Unlike those on most IRC networks, users on the Yahoo! chat network cannot create chat rooms with customized topics because this feature is disabled by Yahoo! to prevent abuses [24]. In addition, users on Yahoo! chat are required to pass a CAPTCHA word verification test in order to join a chat room. This recently-added feature is to guard against a major source of abuse—bots.

### 2.2 Chat Bots

The term *bot*, short for robot, refers to automated programs, that is, programs that do not require a human operator. A chat bot is a program that interacts with a chat service to automate tasks for a human, e.g., creating chat logs. The first-generation chat bots were designed to help operate chat rooms, or to entertain chat users, e.g., quiz or quote bots. However, with the commercialization of the Internet, the main enterprise of chat bots is now sending chat spam. Chat bots deliver spam URLs via either links in chat messages or user profile links. A single bot operator, controlling a few hundred chat bots, can distribute spam links to thousands of users in different chat rooms, making chat bots very profitable to the bot operator who is paid per-click through affiliate programs. Other potential abuses of bots include spreading malware, phishing, booting, and similar malicious activities.

A few countermeasures have been used to defend against the abuse of chat bots, though none of them are very effective. On the server side, CAPTCHA tests are used by Yahoo! chat in an effort to prevent chat bots joining chat rooms. However, this defense becomes ineffective as chat bots bypass CAPTCHA tests with human assistance. We have observed that bots continue to join chat rooms and sometimes even become the majority members of a chat room after the deployment of CAPTCHA tests. Third-party chat clients filter out chat bots, mainly based on key words or key phrases that are known to be used by chat bots. The drawback with this approach is that it cannot capture those unknown or evasive chat bots that do not use the known key words or phrases.

### 2.3 Related Work

Dewes et al. [9] conducted a systematic measurement study of IRC and Web-chat traffic, revealing several statistical properties of chat traffic. (1) Chat sessions tend to last for a long time, and a significant number of IRC sessions last much longer than Web-chat sessions. (2) Chat session inter-arrival time follows an exponential distribution, while the distribution of message inter-arrival time is not exponential. (3) In terms of message size, all chat sessions are dominated by a large number of small packets. (4) Over an entire session, typically a user receives about 10 times as much data as he sends. However, very active users in Web-chat and automated scripts used in IRC may send more data than they receive.

There is considerable overlap between chat and instant messaging (IM) systems, in terms of protocol and user base. Many widely used chat systems such as IRC predate the rise of IM systems, and have great impact upon the IM system and protocol design. In return, some new features that make the IM systems more user-friendly have been back-ported to the chat systems. For example, IRC, a classic chat system, implements a number of IM-like features, such as presence and file transfers, in its current versions. Some messaging service providers, such as Yahoo!, offer both chat and IM accesses to their end-user clients. With this in mind, we outline some related work on IM systems. Liu et al. [21] explored client-side and server-side methods for detecting and filtering IM spam or *spim*. However, their evaluation is based on a corpus of short e-mail spam messages, due to the lack of data on spim. In [23], Mannan et al. studied IM worms, automated malware that spreads on IM systems using the IM contact list. Leveraging the spreading characteristics of IM malware, Xie et al. [41] presented an IM malware detection and suppression system based on the honeypot concept.

Botnets consist of a large number of slave computing assets, which are also called “bots”. However, the usage and behavior of bots in botnets are quite different from those of chat bots. The bots in botnets are malicious programs designed specifically to run on compromised hosts on the Internet, and they are used as platforms to launch a variety of illicit and criminal activities such as credential theft, phishing, distributed denial-of-service attacks, etc. In contrast, chat bots are automated programs designed mainly to interact with chat users by sending spam messages and URLs in chat rooms. Although having been used by botnets as command and control mechanisms [2, 11], IRC and other chat systems do not play an irreplaceable role in botnets. In fact, due to the increasing focus on detecting and thwarting IRC-based botnets [8, 13, 14], recently emerged botnets, such as Phatbot, Nugache, Slapper, and Sinit, show a tendency towards using P2P-based control architectures [39].

Chat spam shares some similarities with email spam. Like email spam, chat spam contains advertisements of illegal services and counterfeit goods, and solicits human users to click spam URLs. Chat bots employ many text obfuscation techniques used by email spam such as word padding and synonym substitution. Since the detection of email spam can be easily converted into the problem of text classification, many content-based filters utilize machine-learning algorithms for filtering email spam. Among them, Bayesian-based statistical approaches [6, 12, 20, 44, 45] have achieved high accuracy and performance. Although very successful, Bayesian-based spam detection techniques still can be evaded by carefully crafted messages [18, 22, 40].

## 3 Measurement

In this section, we detail our measurements on Yahoo! chat, one of the most popular commercial chat services. The focus of our measurements is on public messages posted to Yahoo! chat rooms. The logging of chat messages is available on the standard Yahoo! chat client, as well as most third party chat clients. Upon entering chat, all chat users are shown a disclaimer from Yahoo! that other users can log their messages. However, we consider the contents of the chat logs to be sensitive, so we only present fully-anonymized statistics.

Our data was collected between August and November of 2007. In late August, Yahoo! implemented a CAPTCHA check on entering chat rooms [5, 26], creating technical problems that made their chat rooms unstable for about two weeks [3, 4]. At the same time, Yahoo! implemented a protocol update, preventing most third party chat clients, used by a large proportion of Yahoo! chat users, from accessing the chat rooms. In short, these upgrades made the chat rooms difficult to be accessed for both chat bots and humans. In mid to late September, both chat bot and third party client developers updated their programs. By early October, chat bots were found in Yahoo! chat [25], possibly bypassing the CAPTCHA check with human assistance. Due to these problems and the lack of chat bots in September and early October, we perform our analysis on August and November chat logs. In August and November, we collected a total of 1,440 hours of chat logs. There are 147 individual chat logs from 21 different chat rooms. The process of reading and labeling these chat logs required about 100 hours. To the best of our knowledge, we are the first in the large scale measurement and classification of chat bots.

### 3.1 Log-Based Classification

In order to characterize the behavior of human users and that of chat bots, we need two sets of chat logs pre-labeled as bots and humans. To create such datasets, we perform log-based classification by reading and labeling a large number of chat logs. The chat users are labeled in three categories: human, bot, and ambiguous.

The log-based classification process is a variation of the Turing test. In a standard Turing test [37], the examiner converses with a test subject (a possible machine) for five minutes, and then decides if the subject is a human or a machine. In our classification process, the examiner observes a long conversation between a test subject (a possible chat bot) and one or more third parties, and then decides if the subject is a human or a chat bot. In addition, our examiner checks the content of URLs and typically observes multiple instances of the same chat bot, which further improve our classification accuracy. Moreover, given that the best practice of current artificial intelligences [36] can rarely pass a non-restricted Turing test, our classification of chat bots should be very accurate.

Although a Turing test is subjective, we outline a few important criteria. The main criterion for being labeled as human is a high proportion of specific, intelligent, and human-like responses to other users. In general, if a user's responses suggest more advanced intelligence than current state-of-the-art AI [36], then the user can be labeled as human. The ambiguous label is reserved for non-English, incoherent, or non-communicative users. The criteria for being classified as bot are as follows. The first is the lack of the intelligent responses required for the human label. The second is the repetition of similar phrases either over time or from other users (other instances of the same chat bot). The third is the presence of spam or malware URLs in messages or in the user's profile.

## 3.2 Analysis

In total, our measurements capture 14 different types of chat bots. The different types of chat bots are determined by their triggering mechanisms and text obfuscation schemes. The former relates to message timing, and the latter relates to message content. The two main types of triggering mechanisms observed in our measurements are timer-based and response-based. A timer-based bot sends messages based on a timer, which can be periodic (i.e., fixed time intervals) or random (i.e., variable time intervals). A response-based bot sends messages based on programmed responses to specific content in messages posted by other users.

There are many different kinds of text obfuscation schemes. The purpose of text obfuscation is to vary the content of messages and make bots more difficult to recognize or appear more human-like. We observed four basic text obfuscation methods that chat bots use to evade filtering or detection. First, chat bots introduce random characters or space into their messages, similar to some spam e-mails. Second, chat bots use various synonym phrases to avoid obvious keywords. By this method, a template with several synonyms for multiple words can lead to thousands of possible messages. Third, chat bots use short messages or break up long messages into multiple messages to evade message filters that work on a message-by-message basis. Fourth, and most interestingly, chat bots replay human phrases entered by other chat users.

According to our observation, the main activity of chat bots is to send spam links to chat users. There are two approaches that chat bots use to distribute spam links in chat rooms. The first is to post a message with a spam link directly in the chat room. The second is to enter the spam URL in the chat bot's user profile and then convince the users to view the profile and click the link. Our logs also include some examples of malware spreading via chat rooms. The behavior of malware-spreading chat bots is very similar to that of spam-sending chat bots, as both attempt to lure human users to click links. Although we did not perform detailed malware analysis on links posted in the chat rooms and Yahoo! applies filters to block links to known malicious files, we found several worm instances in our data. There are 12 W32.Imaut.AS [35] worms appeared in the August chat logs, and 23 W32.Imaut.AS worms appeared in the November chat logs. The November worms attempted to send malicious links but were blocked by Yahoo! (the malicious links in their messages being removed), however, the August worms were able to send out malicious links.

The focus of our measurements is mainly on short term statistics, as these statistics are most likely to be useful in chat bot classification. The two key measurement metrics in this study are inter-message delay and message size. Based on these two metrics, we profile the behavior of human and that of chat bots. Among chat bots, we further divide them into four different groups: periodic bots, random bots, responder bots, and replay bots. With respect to these short-term statistics, human and chat bots behave differently, as shown below.

### 3.2.1 Humans

Figure 1 shows the probability distributions of human inter-message delay and message size. Since the behavior of humans is persistent, we only draw the probability mass function (pmf) curves based on the August data. The previous study on Internet chat systems [9] observed that the distribution of inter-message delay in chat systems was heavy tailed. In general our measurement result conforms to that observation. The body part of the pmf curve in Figure 1 (a) (log-log scale) can be linearly fitted, indicating that the distribution of human inter-message delays follows a power law. In other words, the distribution is heavy tailed. We also find that the pmf curve of human message size in Figure 1 (b) can be well fitted by an exponential distribution with $\lambda = 0.034$ after

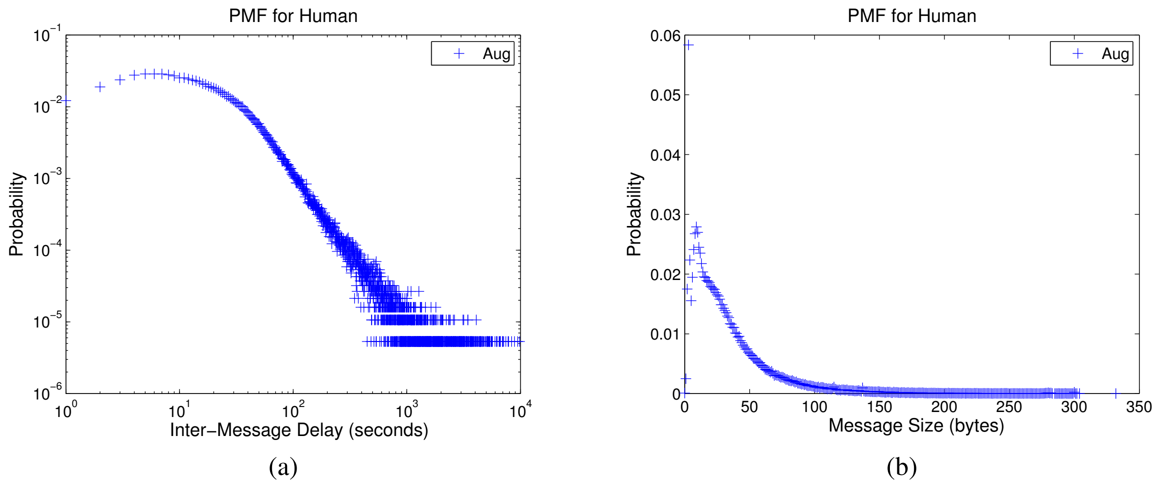


Figure 1: Distribution of human inter-message delay (a) and message size (b)

excluding the initial spike.

### 3.2.2 Periodic Bots

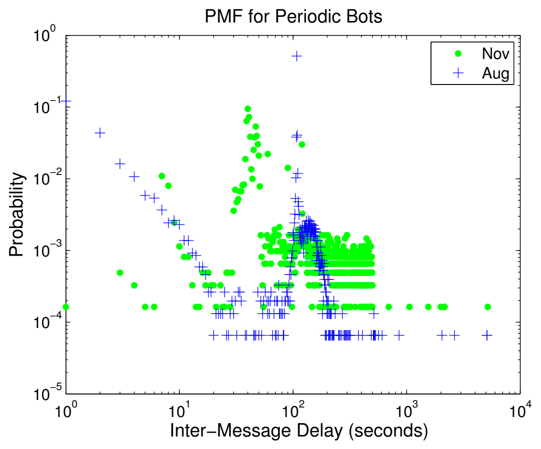
A periodic bot posts messages mainly at regular time intervals. The delay periods of periodic bots, especially those bots that use long delays, may vary by several seconds. The variation of delay period may be attributed to either transmission delay caused by network traffic congestion or chat server delay, or message emission delay incurred by system overloading on the bot hosting machine. The posting of periodic messages is a simple but effective mechanism for distributing messages, so it is not surprising that a substantial portion of chat bots use periodic timers.

We display the probability distributions of inter-message delay and message size for periodic bots in Figure 2. We use '+' for displaying August data and '•' for November data. The distributions of periodic bots are distinct from those of humans shown in Figure 1. The distribution of inter-message delay for periodic bots clearly manifests the timer-triggering characteristic of periodic bots. There are three clusters with high probabilities at time ranges [30-50], [100-110], and [150-170]. These clusters correspond to the November periodic bots with timer values around 40 seconds and the August periodic bots with timer values around 105 and 160 seconds, respectively. The message size pmf curve of the August periodic bots shows an interesting bell shape, much like a normal distribution. After examining message contents, we find that the bell shape may be attributed to the message composition method some August bots used. As shown in Appendix A, some August periodic bots compose a message using a single template. The template has several parts and each part is associated with several synonym phrases. Since the length of each part is independent and identically distributed, the length of whole message, i.e., the sum of all parts, should approximate a normal distribution. The November bots employ a similar composition method, but use several templates of different lengths. Thus, the message size distribution of the November periodic bots reflects the distribution of the lengths of the different templates, with the length of each individual template approximating a normal distribution.

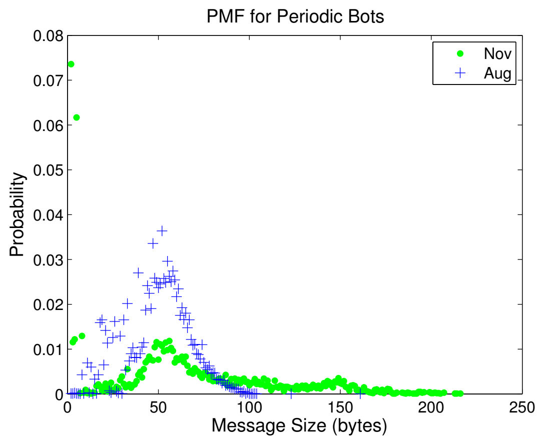
### 3.2.3 Random Bots

A random bot posts messages at random time intervals. The random bots in our data used different random distributions, some discrete and others continuous, to generate inter-message delays. The use of random timers makes random bots appear more human-like than periodic bots. In statistical terms, however, random bots exhibit quite different inter-message delay distributions than humans.

Figure 3 depicts the probability distributions of inter-message delay and message size for random bots. Compared to periodic bots, random bots have more dispersed timer values. In addition, the August random bots have a large overlap with the November random bots. The points with high probabilities (greater than $10^{-2}$) in the time range [30-90] in Figure 3 (a) represent the August and November random bots that use a discrete distribution of 40, 64, and 88 seconds. The wide November cluster with medium probabilities in the time range [40-130] is created by the November random bots that use a uniform distribution between 45 and 125 seconds. The probabilities of different message sizes for the August and November random bots are mainly in the size range [0-50]. Unlike periodic bots, most random bots do not use template or synonym replacement, but directly repeat messages. Thus, as their messages are selected from a database at random, the message size distribution re-

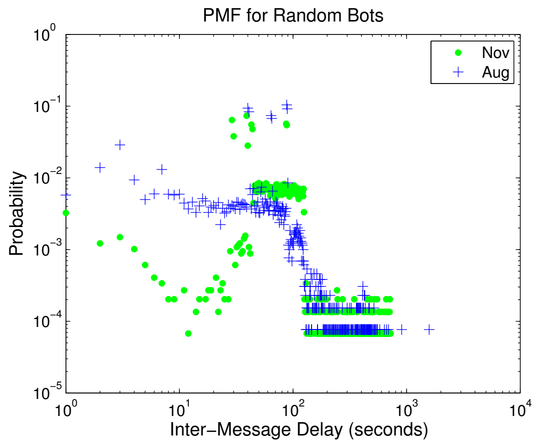


(a)

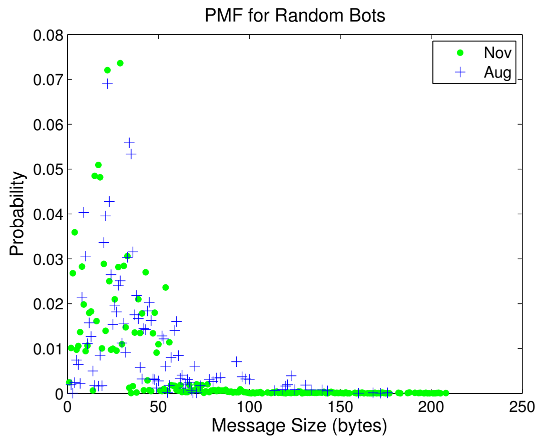


(b)

Figure 2: Distribution of periodic bot inter-message delay (a) and message size (b)



(a)



(b)

Figure 3: Distribution of random bot inter-message delay (a) and message size (b)

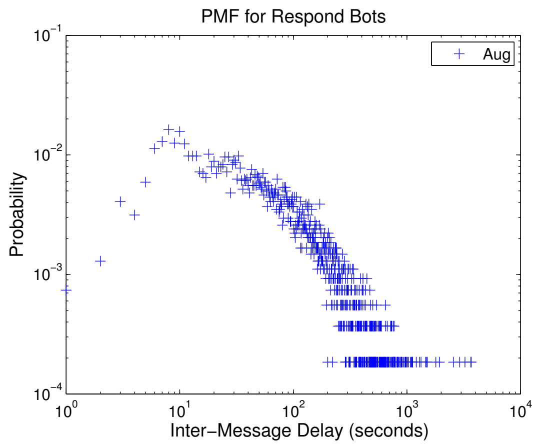
flects the proportion of messages of different sizes in the database.

#### 3.2.4 Responder Bots

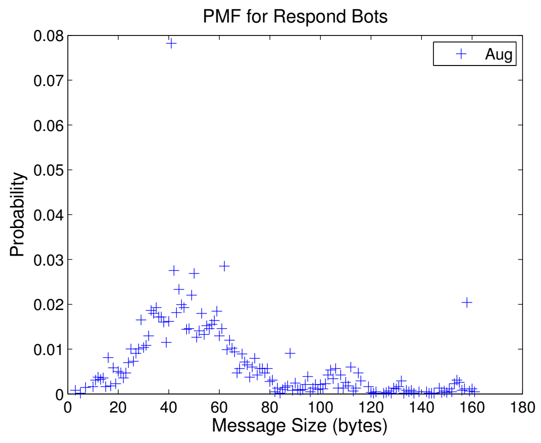
A responder bot sends messages based on the content of messages in the chat room. For example, a message ending with a question mark may trigger a responder bot to send a vague response with a URL, as shown in Appendix A. The vague response, in the context, may trick human users into believing that the responder is a human and further clicking the link. Moreover, the message triggering mechanism makes responder bots look more like humans in terms of timing statistics than periodic or random bots.

To gain more insights into responder bots, we managed to obtain a configuration file for a typical responder bot [38]. There are a number of parameters for making the responder bot mimic humans. The bot can be configured with a fixed typing rate, so that responses with different lengths take different time to "type." The bot can also be set to either ignore triggers while simulating typing, or rate-limit responses. In addition, responses can be assigned with probabilities, so that the responder bot responds to a given trigger in a random manner.

Figure 4 shows the probability distributions of inter-message delay and message size for responder bots. Note that only the distribution of the August responder bots is shown due to the small number of responder bots found in November. Since the message emission of responder bots is triggered by human messages, theoretically the distribution of inter-message delays of responder bots should demonstrate certain similarity to that of humans.

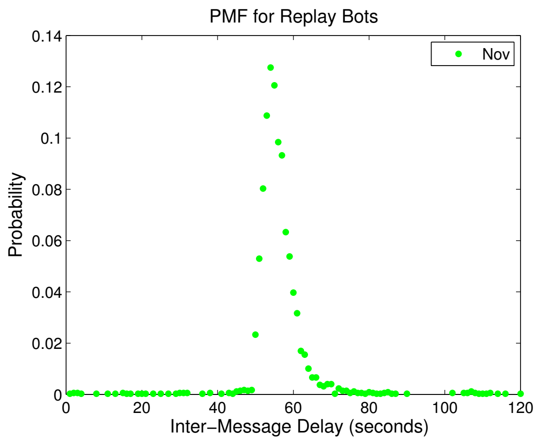


(a)

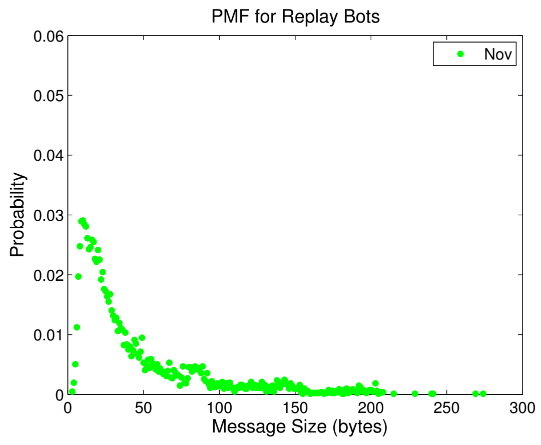


(b)

Figure 4: Distribution of responder bot inter-message delay (a) and message size (b)



(a)



(b)

Figure 5: Distribution of replay bot inter-message delay (a) and message size (b)

Figure 4 (a) confirms this hypothesis. Like Figure 1 (a), the pmf of responder bots (excluding the head part) in log-log scale exhibits a clear sign of a heavy tail. But unlike human messages, the sizes of responder bot messages vary in a much narrower range (between 1 and 160). The bell shape of the distribution for message size less than 100 indicates that responder bots share a similar message composition technique with periodic bots, and their messages are composed as templates with multiple parts, as shown in Appendix A.

### 3.2.5 Replay Bots

A replay bot not only sends its own messages, but also repeats messages from other users to appear more like a human user. In our experience, replayed phrases are related to the same topic but do not appear in the same chat room as the original ones. Therefore, replayed phrases are either taken from other chat rooms on the same topic or saved previously in a database and replayed.

The use of replayed phrases in a crowded or "noisy" chat room does, in fact, make replay bots look more like human to inattentive users. The replayed phrases are sometimes nonsensical in the context of the chat, but human users tend to naturally ignore such statements. When replay bots succeed in fooling human users, these users are more likely to click links posted by the bots or visit their profiles. Interestingly, replay bots sometimes replay phrases uttered by other chat bots, making them very easy to be recognized. The use of replay is potentially effective in thwarting detection methods, as detection tests must deal with a combination of human and bots phrases. By using human phrases, replay bots

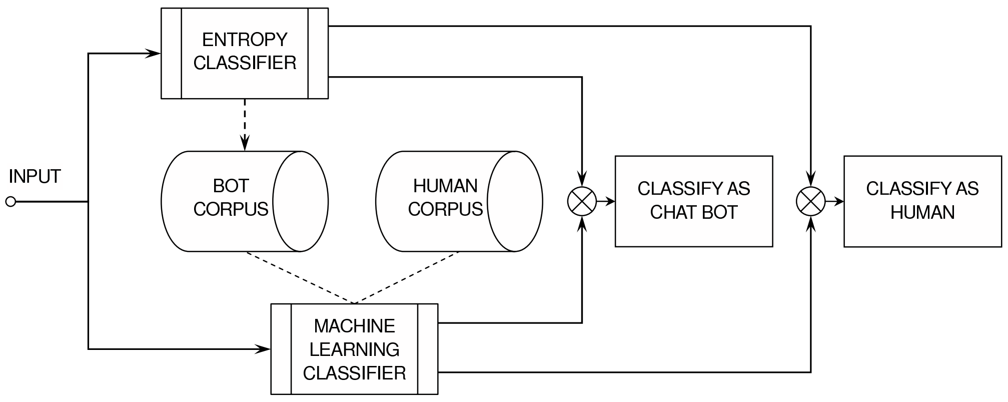


Figure 6: Classification System Diagram

can easily defeat keyword-based message filters that filter message-by-message, as the human phrases should not be filtered out.

Figure 5 illustrates the probability distributions of inter-message delay and message size for replay bots. In terms of inter-message delay, a replay bot is just a variation of a periodic bot, which is demonstrated by the high spike in Figure 5 (a). By using human phrases, replay bots successfully mimic human users in terms of message size distribution. The message size distribution of replay bots in Figure 5 (b) largely resembles that of human users, and can be fitted by an exponential distribution with $\lambda = 0.028$.

## 4 Classification System

This section describes the design of our chat bot classification system. The two main components of our classification system are the entropy classifier and the machine learning classifier. The basic structure of our chat bot classification system is shown in Figure 6. The two classifiers, entropy and machine learning, operate concurrently to process input and make classification decisions, while the machine learning classifier relies on the entropy classifier to build the bot corpus. The entropy classifier uses entropy and corrected conditional entropy to score chat users and then classifies them as chat bots or humans. The main task of the entropy classifier is to capture new chat bots and add them to the chat bot corpus. The human corpus can be taken from a database of clean chat logs or created by manual log-based classification, as described in Section 3. The machine learning classifier uses the bot and human corpora to learn text patterns of bots and humans, and then it can quickly classify chat bots based on these patterns. The two classifiers are detailed as follows.

### 4.1 Entropy Classifier

The entropy classifier makes classification decisions based on entropy and entropy rate measures of message sizes and inter-message delays for chat users. If either the entropy or entropy rate is low for these characteristics, it indicates the regular or predictable behavior of a likely chat bot. If both the entropy and entropy rate is high for these characteristics, it indicates the irregular or unpredictable behavior of a possible human.

To use entropy measures for classification, we set a cutoff score for each entropy measure. If a test score is greater than or equal to the cutoff score, the chat user is classified as a human. If the test score is less than the cutoff score, the chat user is classified as a chat bot. The specific cutoff score is an important parameter in determining the false positive and true positive rates of the entropy classifier. On the one hand, if the cutoff score is too high, then too many humans will be misclassified as bots. On the other hand, if the cutoff score is too low, then too many chat bots will be misclassified as humans. Due to the importance of achieving a low false positive rate, we select the cutoff scores based on human entropy scores to achieve a targeted false positive rate. The specific cutoff scores and targeted false positive rates are described in Section 5.

#### 4.1.1 Entropy Measures

The entropy rate, which is the average entropy per random variable, can be used as a measure of complexity or regularity [10, 30, 31]. The entropy rate is defined as the conditional entropy of a sequence of infinite length. The entropy rate is upper-bounded by the entropy of the first-order probability density function or first-order entropy. A independent and identically distributed (i.i.d.) process has an entropy rate equal to its first-order entropy. A highly complex process has a high entropy rate, while a highly regular process has a low entropy rate.

A random process $X = \{X_i\}$ is defined as an indexed sequence of random variables. To give the definition of the entropy rate of a random process, we first define the entropy of a sequence of random variables as:

$$H(X_1, ..., X_m) = - \sum_{X_1,...,X_m} P(x_1, ..., x_m) \log P(x_1, ..., x_m),$$

where $P(x_1, ..., x_m)$ is the joint probability $P(X_1 = x_1, ..., X_m = x_m)$.

Then, from the entropy of a sequence of random variables, we define the conditional entropy of a random variable given a previous sequence of random variables as:

$$H(X_m \mid X_1, ..., X_{m-1}) = H(X_1, ..., X_m) - H(X_1, ..., X_{m-1}).$$

Lastly, the entropy rate of a random process is defined as:

$$\overline{H}(X) = \lim_{m\to\infty} H(X_m \mid X_1, ..., X_{m-1}).$$

Since the entropy rate is the conditional entropy of a sequence of infinite length, it cannot be measure for finite samples. Thus, we estimate the entropy rate with the conditional entropy of finite samples. In practice, we replace probability density functions with empirical probability density functions based on the method of histograms. The data is binned in $Q$ bins of approximately equal probability. The empirical probability density functions are determined by the proportions of bin number sequences in the data, i.e., the proportion of a sequence is the probability of that sequence. The estimates of the entropy and conditional entropy, based on empirical probability density functions, are represented as: $EN$ and $CE$, respectively.

There is a problem with the estimation of $CE(X_m \mid X_1, ..., X_{m-1})$ for some values of $m$. The conditional entropy tends to zero as $m$ increases, due to limited data. If a specific sequence of length $m-1$ is found only once in the data, then the extension of this sequence to length $m$ will also be found only once. Therefore, the length $m$ sequence can be predicted by the length $m-1$ sequence, and the length $m$ and $m-1$ sequences cancel out. If no sequence of length $m$ is repeated in the data, then $CE(X_m \mid X_1, ..., X_{m-1})$ is zero, even for i.i.d. processes.

To solve the problem of limited data, without fixing the length of $m$, we use the corrected conditional entropy [30] represented as $CCE$. The corrected conditional entropy is defined as:

$$CCE(X_m \mid X_1, ..., X_{m-1}) = CE(X_m \mid X_1, ..., X_{m-1}) + perc(X_m) \cdot EN(X_1),$$

where $perc(X_m)$ is the percentage of unique sequences of length $m$ and $EN(X_1)$ is the entropy with $m$ fixed at 1 or the first-order entropy.

The estimate of the entropy rate is the minimum of the corrected conditional entropy over different values of $m$. The minimum of the corrected conditional entropy is considered to be the best estimate of the entropy rate from the available data.

## 4.2 Machine Learning Classifier

The machine learning classifier uses the content of chat messages to identify chat bots. Since chat messages (including emoticons) are text, the identification of chat bots can be perfectly fitted into the domain of machine learning text classification. Within the machine learning paradigm, the text classification problem can be formalized as $f : T \times C \to \{0, 1\}$, where $f$ is the classifier, $T = \{t_1, t_2, ..., t_n\}$ is the texts to be classified, and $C = \{c_1, c_2, ..., c_k\}$ is the set of pre-defined classes [33]. Value 1 for $f(t_i, c_j)$ indicates that text $t_i$ is in class $c_j$ and value 0 indicates the opposite decision. There are many techniques that can be used for text classification, such as naïve Bayes, support vector machines, and decision trees. Among them, Bayesian classifiers have been very successful in text classification, particularly in email spam detection. Due to the similarity between chat spam and email spam, we choose Bayesian classification for our machine learning classifier for detecting chat bots. We leave study on the applicability of other types of machine learning classifiers to our future work.

Within the framework of Bayesian classification, identifying if chat message $M$ is issued by a bot or human is achieved by computing the probability of $M$ being from a bot with the given message content, i.e., $P(C = bot|M)$. If the probability is equal to or greater than a pre-defined threshold, then message $M$ is classified as a bot message. According to Bayes theorem,

$$P(bot|M) = \frac{P(M|bot)P(bot)}{P(M)} = \frac{P(M|bot)P(bot)}{P(M|bot)P(bot) + P(M|human)P(human)}.$$

A message $M$ is described by its feature vector $\langle f_1, f_2, ..., f_n \rangle$. A feature $f$ is a single word or a combination of multiple words in the message. To simplify computation, in practice it is usually assumed that all features are conditionally independent with each other for

Table 1: Message Composition of Chat Bot and Human Datasets

| | AUG. BOTS | | | NOV. BOTS | | | HUMANS |
|---|---|---|---|---|---|---|---|
| | periodic | random | responder | periodic | random | replay | human |
| number of messages | 25,258 | 13,998 | 6,160 | 10,639 | 22,820 | 8,054 | 342,696 |

the given category. Thus, we have

$$P(bot|M) = \frac{P(bot) \prod_{i=1}^{n} P(f_i|bot)}{P(bot) \prod_{i=1}^{n} P(f_i|bot) + P(human) \prod_{i=1}^{n} P(f_i|human)}.$$

The value of $P(bot|M)$ may vary in different implementations (see [12, 45] for implementation details) of Bayesian classification due to differences in assumption and simplification.

Given the abundance of implementations of Bayesian classification, we directly adopt one implementation, namely CRM 114 [44], as our machine learning classification component. CRM 114 is a powerful text classification system that has achieved very high accuracy in email spam identification. The default classifier of CRM 114, OSB (Orthogonal Sparse Bigram), is a type of Bayesian classifier. Different from common Bayesian classifiers which treat individual words as features, OSB uses word pairs as features instead. OSB first chops the whole input into multiple basic units with five consecutive words in each unit. Then, it extracts four word pairs from each unit to construct features, and derives their probabilities. Finally, OSB applies Bayes theorem to compute the overall probability that the text belongs to one class or another.

## 5 Experimental Evaluation

In this section, we evaluate the effectiveness of our proposed classification system. Our classification tests are based on chat logs collected from the Yahoo! chat system. We test the two classifiers, entropy-based and machine-learning-based, against chat bots from August and November datasets. The machine learning classifier is tested with fully-supervised training and entropy-classifier-based training. The accuracy of classification is measured in terms of false positive and false negative rates. The false positives are those human users that are misclassified as chat bots, while the false negatives are those chat bots that are misclassified as human users. The speed of classification is mainly determined by the minimum number of messages that are required for accurate classification. In general, a high number means slow classification, whereas a low number means fast classification.

### 5.1 Experimental Setup

The chat logs used in our experiments are mainly in three datasets: (1) human chat logs from August 2007, (2) bot chat logs from August 2007, and (3) bot chat logs from November 2007. In total, these chat logs contain 342,696 human messages and 87,049 bot messages. In our experiments, we use the first half of each chat log, human and bot, for training our classifiers and the second half for testing our classifiers. The composition of the chat logs for the three datasets is listed in Table 1.

The entropy classifier only requires a human training set. We use the human training set to determine the cutoff scores, which are used by the entropy classifier to decide whether a test sample is a human or bot. The target false positive rate is set at 0.01. To achieve this false positive rate, the cutoff scores are set at approximately the 1st percentile of human training set scores. Then, samples that score higher than the cutoff are classified as humans, while samples that score lower than the cutoff are classified as bots. The entropy classifier uses two entropy tests: entropy and corrected conditional entropy. The entropy test estimates first-order entropy, and the corrected conditional entropy estimates higher-order entropy or entropy rate. The corrected conditional entropy test is more precise with coarse-grain bins, whereas the entropy test is more accurate with fine-grains bins [10]. Therefore, we use $Q = 5$ for the corrected conditional entropy test and $Q = 256$ with $m$ fixed at 1 for the entropy test.

We run classification tests for each bot type using the entropy classifier and machine learning classifier. The machine learning classifier is tested based on fully-supervised training and then entropy-based training. In fully-supervised training, the machine learning classifier is trained with manually labeled data, as described in Section 3. In entropy-based training, the machine learning classifier is trained with data labeled by the entropy classifier. For each evaluation, the entropy classifier uses samples of 100 messages, while the machine learning classifier uses samples of 25 messages.

### 5.2 Experimental Results

We now present the results for the entropy classifier and machine learning classifier. The four chat bot types are: periodic, random, responder, and replay. The classification tests are organized by chat bot type, and are ordered by increasing detection difficulty.

Table 2: Entropy Classifier Accuracy

| | AUG. BOTS | | | NOV. BOTS | | | HUMANS |
|---|---|---|---|---|---|---|---|
| | periodic | random | responder | periodic | random | replay | human |
| test | true pos. | true pos. | true pos. | true pos. | true pos. | true pos. | false pos. |
| EN(imd) | 121/121 | 68/68 | 1/30 | 51/51 | 109/109 | 40/40 | 7/1713 |
| CCE(imd) | 121/121 | 49/68 | 4/30 | 51/51 | 109/109 | 40/40 | 11/1713 |
| EN(ms) | 92/121 | 7/68 | 8/30 | 46/51 | 34/109 | 0/40 | 7/1713 |
| CCE(ms) | 77/121 | 8/68 | 30/30 | 51/51 | 6/109 | 0/40 | 11/1713 |
| OVERALL | 121/121 | 68/68 | 30/30 | 51/51 | 109/109 | 40/40 | 17/1713 |

#### 5.2.1 Entropy Classifier

The detection results of the entropy classifier are listed in Table 2, which includes the results of the entropy test ($EN$) and corrected conditional entropy test ($CCE$) for inter-message delay ($imd$), and message size ($ms$). The overall results for all entropy-based tests are shown in the final row of the table. The true positives are the total unique bot samples correctly classified as bots. The false positives are the total unique human samples mistakenly classified as bots.

**Periodic Bots**: As the simplest group of bots, periodic bots are the easiest to detect. They use different fixed timers and repeatedly post messages at regular intervals. Therefore, their inter-message delays are concentrated in a narrower range than those of humans, resulting in lower entropy than that of humans. The inter-message delay $EN$ and $CCE$ tests detect 100% of all periodic bots in both August and November datasets. The message size $EN$ and $CCE$ tests detect 76% and 63% of the August periodic bots, respectively, and 90% and 100% of the November periodic bots, respectively. These slightly lower detection rates are due to a small proportion of humans with low entropy scores that overlap with some periodic bots. These humans post mainly short messages, resulting in message size distributions with low entropy.

**Random Bots**: The random bots use random timers with different distributions. Some random bots use discrete timings, e.g., 40, 64, or 88 seconds, while the others use continuous timings, e.g., uniformly distributed delays between 45 and 125 seconds.

The inter-message delay $EN$ and $CCE$ tests detect 100% of all random bots, with one exception: the inter-message delay $CCE$ test against the August random bots only achieves 72% detection rate, which is caused by the following two conditions: (1) the range of message delays of random bots is close to that of humans; (2) sometimes the randomly-generated delay sequences have similar entropy rate to human patterns. The message size $EN$ and $CCE$ tests detect 31% and 6% of August random bots, respectively, and 7% and 8% of November random bots, respectively. These low detection rates are again due to a small proportion of humans with low message size entropy scores. However, unlike periodic bots, the message size distribution of random bots is highly dispersed, and thus, a larger proportion of random bots have high entropy scores, which overlap with those of humans.

**Responder Bots**: The responder bots are among the advanced bots, and they behave more like humans than random or periodic bots. They are triggered to post messages by certain human phrases. As a result, their timings are quite similar to those of humans.

The inter-message delay $EN$ and $CCE$ tests detect very few responder bots, only $3\%$ and $13\%$, respectively. This demonstrates that human-message-triggered responding is a simple yet very effective mechanism for imitating the timing of human interactions. However, the detection rate for the message size $EN$ test is slightly better at $27\%$, and the detection rate for the message size $CCE$ test reaches $100\%$. While the message size distribution has sufficiently high entropy to frequently evade the $EN$ test, there is some dependence between subsequent message sizes, and thus, the $CCE$ detects the low entropy pattern over time.

**Replay Bots**: The replay bots also belong to the advanced and human-like bots. They use replay attacks to fool humans. More specifically, the bots replay phrases they observed in chat rooms. Although not sophisticated in terms of implementation, the replay bots are quite effective in deceiving humans as well as frustrating our message-size-based detections: the message size $EN$ and $CCE$ tests both have detection rates of $0\%$. Despite their clever trick, the timing of replay bots is periodic and easily detected. The inter-message delay $EN$ and $CCE$ tests are very successful at detecting replay bots, both with 100% detection accuracy.

#### 5.2.2 Supervised and Hybrid Machine Learning Classifiers

The detection results of the machine learning classifier are listed in Table 3. Table 3 shows the results for the fully-supervised machine learning ($SupML$) classifier and entropy-trained machine learning ($EntML$) classifier, both trained on the August training datasets, and the

Table 3: Machine Learning Classifier Accuracy

| | AUG. BOTS | | | NOV. BOTS | | | HUMANS |
|---|---|---|---|---|---|---|---|
| | periodic | random | responder | periodic | random | replay | human |
| test | true pos. | true pos. | true pos. | true pos. | true pos. | true pos. | false pos. |
| $SupML$ | 121/121 | 68/68 | 30/30 | 14/51 | 104/109 | 1/40 | 0/1713 |
| $SupMLretrained$ | 121/121 | 68/68 | 30/30 | 51/51 | 109/109 | 40/40 | 0/1713 |
| $EntML$ | 121/121 | 68/68 | 30/30 | 51/51 | 109/109 | 40/40 | 1/1713 |

fully-supervised machine learning ($SupMLretrained$) classifier trained on August and November training datasets.

**Periodic Bots**: For the August dataset, both $SupML$ and $EntML$ classifiers detect 100% of all periodic bots. For the November dataset, however, the $SupML$ classifier only detects 27% of all periodic bots. The lower detection rate is due to the fact that 62% of the periodic bot messages in November chat logs are generated by new bots, making the $SupML$ classifier ineffective without re-training. The $SupMLretrained$ classifier detects 100% of November periodic bots. The $EntML$ classifier also achieves 100% for the November dataset.

**Random Bots**: For the August dataset, both $SupML$ and $EntML$ classifiers detect 100% of all random bots. For the November dataset, the $SupML$ classifier detects 95% of all random bots, and the $SupMLretrained$ classifier detects 100% of all random bots. Although 52% of the random bots have been upgraded according to our observation, the old training set is still mostly effective because certain content features of August random bots still appear in November. The $EntML$ classifier again achieves 100% detection accuracy for the November dataset.

**Responder Bots**: We only present the detection results of responder bots for the August dataset, as the number of responder bots in the November dataset is very small. Although responder bots effectively mimic human timing, their message contents are only slightly obfuscated and are easily detected. The $SupML$ and $EntML$ classifiers both detect 100% of all responder bots.

**Replay Bots**: The replay bots only exist in the November dataset. The $SupML$ classifier detects only 3% of all replay bots, as these bots are newly introduced in November. The $SupMLretrained$ classifier detects 100% of all replay bots. The machine learning classifier reliably detects replay bots in the presence of a substantial number of replayed human phrases, indicating the effectiveness of machine learning techniques in chat bot classification.

## 6 Conclusion and Future Work

This paper first presents a large-scale measurement study on Internet chat. We collected two-month chat logs for 21 different chat rooms from one of the top Internet chat service providers. From the chat logs, we identified a total of 14 different types of chat bots and grouped them into four categories: periodic bots, random bots, responder bots, and replay bots. Through statistical analysis on inter-message delay and message size for both chat bots and humans, we found that chat bots behave very differently from human users. More specifically, chat bots exhibit certain regularities in either inter-message delay or message size. Although responder bots and replay bots employ advanced techniques to behave more human-like in some aspects, they still lack the overall sophistication of humans.

Based on the measurement study, we further proposed a chat bot classification system, which utilizes entropy-based and machine-learning-based classifiers to accurately detect chat bots. The entropy-based classifier exploits the low entropy characteristic of chat bots in either inter-message delay or message size, while the machine-learning-based classifier leverages the message content difference between humans and chat bots. The entropy-based classifier is able to detect unknown bots, including human-like bots such as responder and replay bots. However, it takes a relatively long time for detection, i.e., a large number of messages are required. Compared to the entropy-based classifier, the machine-learning-based classifier is much faster, i.e., a small number of messages are required. In addition to bot detection, a major task of the entropy-based classifier is to build and maintain the bot corpus. With the help of bot corpus, the machine-learning-based classifier is trained, and consequently, is able to detect chat bots quickly and accurately. Our experimental results demonstrate that the hybrid classification system is fast in detecting known bots and is accurate in identifying previously-unknown bots.

There are a number of possible directions for our future work. We plan to explore the application of entropy-based techniques in detecting other forms of bots, such as web bots. We also plan to investigate the development of more advanced chat bots that could evade our hybrid classification system. We believe that the continued work in this area will reveal other important characteristics of bots and automated programs, which is useful in malware detection and prevention.

## Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] AHN, L. V., BLUM, M., HOPPER, N., AND LANGFORD, J. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt* (Warsaw, Poland, May 2003).

[2] BÄCHER, P., HOLZ, T., KÖTTER, M., AND WICHERSKI, G. Know your enemy: Tracking botnets, 2005. `http://www.honeynet.org/papers/bots` [Accessed: Jan. 25, 2008].

[3] BACON, S. Chat rooms follow-up. `http://www.ymessengerblog.com/blog/2007/08/21/chat-rooms-follow-up/` [Accessed: Jan. 25, 2008].

[4] BACON, S. Chat rooms update. `http://www.ymessengerblog.com/blog/2007/08/24/chat-rooms-update-2/` [Accessed: Jan. 25, 2008].

[5] BACON, S. New entry process for chat rooms. `http://www.ymessengerblog.com/blog/2007/08/29/new-entry-process-for-cha%t-rooms/` [Accessed: Jan. 25, 2008].

[6] BLOSSER, J., AND JOSEPHSEN, D. Scalable centralized bayesian spam mitigation with bogofilter. In *Proceedings of the 2004 USENIX Systems Administration Conference (LISA'04)* (Atlanta, GA., USA, November 2004).

[7] CRISLIP, D. Will Blizzard's spam-stopper really work? `http://www.wowinsider.com/2007/05/16/will-blizzards-spam-stopper-really-work/` [Accessed: Dec. 25, 2007].

[8] DAGON, D., GU, G., LEE, C. P., AND LEE, W. A taxonomy of botnet structures. In *Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC'07)* (Miami, FL., USA, December 2007).

[9] DEWES, C., WICHMANN, A., AND FELDMANN, A. An analysis of Internet chat systems. In *Proceedings of the 2003 ACM/SIGCOMM Internet Measurement Conference (IMC'03)* (Miami, FL., USA, October 2003).

[10] GIANVECCHIO, S., AND WANG, H. Detecting covert timing channels: An entropy-based approach. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS'07)* (Alexandria, VA., USA, October 2007).

[11] GOEBEL, J., AND HOLZ, T. Rishi: Identify bot contaminated hosts by IRC nickname evaluation. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnets (HotBots'07)* (Cambridge, MA., USA, April 2007).

[12] GRAHAM, P. A plan for spam, 2002. `http://www.paulgraham.com/spam.html` [Accessed: Jan. 25, 2008].

[13] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. Bothunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 2007 USENIX Security Symposium (Security'07)* (Boston, MA., USA, August 2007).

[14] GU, G., ZHANG, J., AND LEE, W. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 2008 Annual Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA., USA, February 2008).

[15] HU, J. AOL: spam and chat don't mix. `http://www.news.com/AOL-Spam-and-chat-dont-mix/2100-1032_3-1024010.html` [Accessed: Jan. 7, 2008].

[16] HU, J. Shutting of MSN chat rooms may open up IM. `http://www.news.com/Shutting-of-MSN-chat-rooms-may-open-up-IM/2100-1025_3-5082677.html` [Accessed: Jan. 7, 2008].

[17] JENNINGS III, R. B., NAHUM, E. M., OLSHEFSKI, D. P., SAHA, D., SHAE, Z.-Y., AND WATERS, C. A study of internet instant messaging and chat protocols. *IEEE Network Vol. 20*, No. 4 (2006), 16–21.

[18] KARLBERGER, C., BAYLER, G., KRUEGEL, C., AND KIRDA, E. Exploiting redundancy in natural language to penetrate bayesian spam filters. In *Proceedings of the USENIX Workshop on Offensive Technologies* (Boston, MA., USA, August 2007).

[19] KREBS, B. Yahoo! messenger network overrun by bots. `http://blog.washingtonpost.com/securityfix/2007/08/yahoo_messenger_network_overru.html` [Accessed: Dec. 18, 2007].

[20] LI, K., AND ZHONG, Z. Fast statistical spam filter by approximate classifications. In *Proceedings of 2006 ACM/SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (St. Malo, France, June 2006).

[21] LIU, Z., LIN, W., LI, N., AND LEE, D. Detecting and filtering instant messaging spam - a global and personalized approach. In *Proceedings of the IEEE Workshop on Secure Network Protocols (NPSEC'05)* (Boston, MA., USA, November 2005).

[22] LOWD, D., AND MEEK, C. Good word attacks on statistical spam filters. In *Proceedings of the 2005 Conference on Email and Anti-Spam (CEAS'05)* (Mountain View, CA., USA, July 2005).

[23] MANNAN, M., AND VAN OORSCHOT, P. C. On instant messaging worms, analysis and countermeasures. In *Proceedings of the ACM Workshop on Rapid Malcode* (Fairfax, VA., USA, November 2005).

[24] MILLS, E. Yahoo! closes chat rooms over child sex concerns. `http://www.news.com/Yahoo-closes-chat-rooms-over-/child-sex-concerns/2100-1025_3-5759705.html` [Accessed: Jan. 27, 2008].

[25] MOHTA, A. Bots are back in Yahoo! chat rooms. `http://www.technospot.net/blogs/bots-are-back-in-yahoo-chat-room/` [Accessed: Dec. 18, 2007].

[26] MOHTA, A. Yahoo! chat adds CAPTCHA check to remove bots. `http://www.technospot.net/blogs/yahoo-chat-captcha-check-to-remove-bots/` [Accessed: Dec. 18, 2007].

[27] NINO, T. Linden Lab taking action against landbots. `http://www.secondlifeinsider.com/2007/05/18/linden-lab-taking-action-against-landbots/` [Accessed: Jan. 7, 2008].

[28] PETITION ONLINE. Action against the Yahoo! bot problem petition. `http://www.petitiononline.com/` [Accessed: Dec. 18, 2007].

[29] PETITION ONLINE. AOL no more chat room spam petition. `http://www.petitiononline.com/` [Accessed: Dec. 18, 2007].

[30] PORTA, A., BASELLI, G., LIBERATI, D., MONTANO, N., COGLIATI, C., GNECCHI-RUSCONE, T., MALLIANI, A., AND CERUTTI, S. Measuring regularity by means of a corrected conditional entropy in sympathetic outflow. *Biological Cybernetics Vol. 78*, No. 1 (January 1998).

[31] ROSIPAL, R. *Kernel-Based Regression and Objective Nonlinear Measures to Assess Brain Functioning*. PhD thesis, University of Paisley, Paisley, Scotland, UK, September 2001.

[32] SCHRAMM, M. Chat spam measures shut down multi-line reporting add-ons. `http://www.wowinsider.com/2007/10/25/chat-spam-measures-shut-down-multi-line-reporting-addons/` [Accessed: Jan. 17, 2008].

[33] SEBASTIANI, F. Machine learning in automated text categorization. *ACM Computing Surveys Vol. 34*, No. 1 (2002), 1–47.

[34] SIMPSON, C. Yahoo! chat anti-spam resource center. `http://www.chatspam.org/` [Accessed: Sep. 25, 2007].

[35] SYMANTEC SECURITY RESPONSE. W32.Imaut.AS worm. `http://www.symantec.com/security_response/writeup.jsp?docid=2007-080114-2713-99` [Accessed: Jan. 25, 2008].

[36] THE ALICE ARTIFICIAL INTELLIGENCE FOUNDATION. ALICE(Artificial Linguistice Internet Computer Entity). `http://www.alicebot.org/` [Accessed: Jan. 25, 2008].

[37] TURING, A. M. Computing machinery and intelligence. *Mind Vol. 59* (1950), 433–460.

[38] UBER-GEEK.COM. Yahoo! responder bot. `http://www.uber-geek.com/bot.html` [Accessed: Jan. 18, 2008].

[39] WANG, P., SPARKS, S., AND ZOU, C. C. An advanced hybrid peer-to-peer botnet. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnets (HotBots'05)* (Cambridge, MA., USA, April 2007).

[40] WITTEL, G. L., AND WU, S. F. On attacking statistical spam filters. In *Proceedings of the 2004 Conference on Email and Anti-Spam (CEAS'04)* (Mountain View, CA., USA, July 2004).

[41] XIE, M., WU, Z., AND WANG, H. HoneyIM: Fast detection and suppression of instant messaging malware in enterprise-like networks. In *Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC'07)* (Miami Beach, FL, USA, December 2007).

[42] YAHELITE.ORG. Yahelite chat client. `http://www.yahelite.org/` [Accessed: Jan. 8, 2008].

[43] YAZAKPRO.COM. Yazak pro chat client. `http://www.yazakpro.com/` [Accessed: Jan. 8, 2008].

[44] YERAZUNIS, B. CRM114 - the controllable regex mutilator, 2003. `http://crm114.sourceforge.net` [Accessed: Jan. 25, 2008].

[45] ZDZIARSKI, J. A. *Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification*. No Starch Press, 2005.

## A Chat Bot Examples

Note that in a chat room the following example messages would be spread out over several minutes.

Example 1: Response Template

```
bot: user1, that's a damn good question.
bot: user1, To know more about Seventh-day Adventist; visit http://www.sda.org
Sabbath; http://www.sabbathtruth.com EGW; http://www.whiteestate.org
bot: user2, no! don't leave me.

bot: user1, too much coffee tonight?
bot: user2, boy, you're just full of questions, aren't you?
bot: user2, lots of evidence for evolution can be found here http://www.talk
origins.org/faqs/comdesc/
```

In the above example, the bot uses a template with three parts to post links:
[username], [link description phrase] [link].

Example 2: Synonym Template

```
bot: Allo Hunks! Enjoy Marjorie! Check My Free Pics
bot: What's happening Guys! Marjorie Here! See more of me at My Free Pics
bot: Hi Babes! I am Marjorie! Rate My Live Cam
bot: Horny lover Guys! Marjorie at your service! Inspect My Site
bot: Mmmm Folks! Im Marjorie! View My Webpage
```

In the above example, the bot uses a template with three parts to post messages:
[salutation phrase]! [introduction phrase]! [web site advertisement phrase].

Example 3: Character Padding

```
bot: anyone boredjn wanna chat?uklcss
bot: any guystfrom the US/Canada hereiqjss
bot: hiyafxqss
bot: ne1 hereqbored?fiqss
bot: ne guysmwanna chat? ciuneed some1 to make megsmile :-)pktpss
```

In the above example, the bot adds random characters to messages.

# *To Catch a Predator*: A Natural Language Approach for Eliciting Malicious Payloads

Sam Small
*Johns Hopkins University*

Joshua Mason
*Johns Hopkins University*

Fabian Monrose
*Johns Hopkins University*

Niels Provos
*Google Inc.*

Adam Stubblefield
*Johns Hopkins University*

## Abstract

We present an automated, scalable, method for crafting dynamic responses to real-time network requests. Specifically, we provide a flexible technique based on natural language processing and string alignment techniques for intelligently interacting with protocols trained directly from raw network traffic. We demonstrate the utility of our approach by creating a low-interaction web-based honeypot capable of luring attacks from search worms targeting hundreds of different web applications. In just over two months, we witnessed over $368,000$ attacks from more than $5,600$ botnets targeting several hundred distinct webapps. The observed attacks included several exploits detected the same day the vulnerabilities were publicly disclosed. Our analysis of the payloads of these attacks reveals the state of the art in search-worm based botnets, packed with surprisingly modular and diverse functionality.

## 1 Introduction

Automated network attacks by malware pose a significant threat to the security of the Internet. Nowadays, web servers are quickly becoming a popular target for exploitation, primarily because once compromised, they open new avenues for infecting vulnerable clients that subsequently visit these sites. Moreover, because web servers are generally hosted on machines with significant system resources and network connectivity, they can serve as reliable platforms for hosting malware (particularly in the case of server farms), and as such, are enticing targets for attackers [25]. Indeed, lately we have witnessed a marked increase in so-called "search worms" that seek out potential victims by crawling the results returned by malevolent search-engine queries [24, 28]. While this new change in the playing field has been noted for some time now, little is known about the scope of this growing problem.

To better understand this new threat, researchers and practitioners alike have recently started to move towards the development of low-interaction, *web-based* honeypots [3]. These differ from traditional honeypots in that their only purpose is to monitor automated attacks directed at vulnerable web applications. However, web-based honeypots face a unique challenge—they are ineffective if not broadly indexed under the same queries used by malware to identify vulnerable hosts. At the same time, the large number of different web applications being attacked poses a daunting challenge, and the sheer volume of attacks calls for efficient solutions. Unfortunately, current web-based honeypot projects tend to be limited in their ability to easily simulate diverse classes of vulnerabilities, require non-trivial amounts of manual support, or do not scale well enough to meet this challenge.

A fundamental difference between the type of malware captured by traditional honeypots (e.g., Honeyd [23]) and approaches geared towards eliciting payloads from search-based malware stems from how potential victims are targeted. For traditional honeypots, these systems can be deployed at a network telescope [22], for example, and can simply take advantage of the fact that for random scanning malware, any traffic that reaches the telescope is unsolicited and likely malicious in nature. However, search-worms use a technique more akin to instantaneous hit-list automation, thereby only targeting authentic and vulnerable hosts. Were web-based honeypots to mimic the passive approach used for traditional honeypots, they would likely be very ineffective.

To address these limitations, we present a method for crafting dynamic responses to on-line network requests using sample transcripts from observed network interaction. In particular, we provide a flexible technique based on natural language processing and string alignment techniques for intelligently interacting with protocols trained directly from raw traffic. Though our approach is application-agnostic, we demonstrate its utility with a system designed to monitor and capture automated network attacks against vulnerable web applications, without relying on static vulnerability signatures. Specifically, our approach (disguised as a typical web server) elicits interaction with search engines and, in turn, search worms in the hope of capturing their illicit payload. As we show later, our dynamic content generation technique is fairly robust and easy to deploy. Over a 72-day period we were attacked repeatedly, and witnessed more than 368,000 attacks originating from 28,856 distinct IP addresses.

The attacks target a wide range of web applications, many of which attempt to exploit the vulnerable application(s) via a diverse set of injection techniques. To our surprise, even during this short deployment phase, we witnessed several attacks immediately after public disclosure of the vulnerabilities being exploited. That, by itself, validates our technique and underscores both the tenacity of attackers and the overall pervasiveness of web-based exploitation. Moreover, the relentless nature of these attacks certainly sheds light on the scope of this problem, and calls for immediate solutions to better curtail this increasing threat to the security of the Internet. Lastly, our forensic analysis of the captured payloads confirms several earlier findings in the literature, as well as highlights some interesting insights on the post-infection process and the malware themselves.

The rest of the paper is organized as follows. Section 2 discusses related work. We provide a high-level overview of our approach in Section 3, followed by specifics of our generation technique in Section 4. We provide a validation of our approach based on interaction with a rigid binary protocol in Section 5. Additionally, we present our real-world deployment and discuss our findings in Section 6. Finally, we conclude in Section 7.

## 2 Related Work

Generally speaking, honeypots are deployed with the intention of eliciting interaction from unsuspecting adversaries. The utility in capturing this interaction has been diverse, allowing researchers to discover new patterns and trends in malware propagation [28], generate new signatures for intrusion-detection systems and Internet security software [16, 20, 31], collect malware binaries for static and/or dynamic analysis [21], and quantify malicious behavior through widespread measurement studies [26], to name a few.

The adoption of virtual honeypots by the security community only gained significant traction after the introduction of *low-interaction* honeypots such as *Honeyd* [23]. Honeyd is a popular tool for establishing multiple virtual hosts on a single machine. Though Honeyd has proved to be fairly useful in practice, it is important to recognize that its effectiveness is strictly tied to the availability of accurate and representative protocol-emulation scripts, whose generation can be fairly tedious and time consuming. *High-interaction* honeypots use a different approach, replying with authentic and unscripted responses by hosting sand-boxed virtual machines running common software and operating systems [11][1].

A number of solutions have been proposed to bridge the separation of benefits and restrictions that exist between high and low-interaction honeypots. For example, Leita et al. proposed *ScriptGen* [18, 17], a tool that automatically generates Honeyd scripts from network traffic logs. ScriptGen creates a finite state machine for each listening port. Unfortunately, as the amount and diversity of available training data grows, so does the size and complexity of its state machines. Similarly, RolePlayer (and its successor, GQ [10]) generates scripts capable of interacting with live traffic (in particular, worms) by analyzing series of similar application sessions to determine static and dynamic fields and then replay appropriate responses. This is achieved by using a number of heuristics to remove common contextual values from annotated traffic samples and using byte-sequence alignment to find potential session identifiers and length fields.

While neither of these systems specifically target search-based malware, they represent germane approaches and many of the secondary techniques they introduce apply to our design as well. Also, their respective designs illustrate an important observation—the choice between using a small or large set of sample data manifests itself as a system tradeoff: there is little diversity to the requests recognized and responses transmitted by RolePlayer, thereby limiting its ability to interact with participants whose behavior deviates from the training session(s). On the other hand, the flexibility provided by greater state coverage in ScriptGen comes at a cost to scalability and complexity.

Lastly, since web-based honeypots rely on search engines to index their attack signatures, they are at a disadvantage each time a new attack emerges. In our work, we sidestep the indexing limitations common to static signature web-based honeypots and achieve broad query representation prior to new attacks by proactively generating "signatures" using statistical language models trained on common web-application scripts. When indexed, these signatures allow us to monitor attack behavior conducted by search worms without explicitly deploying structured signatures *a priori*.

## 3 High-level Overview

We now briefly describe our system architecture. Its setup follows the description depicted in Figure 1, which is conceptually broken into three stages: pre-processing,

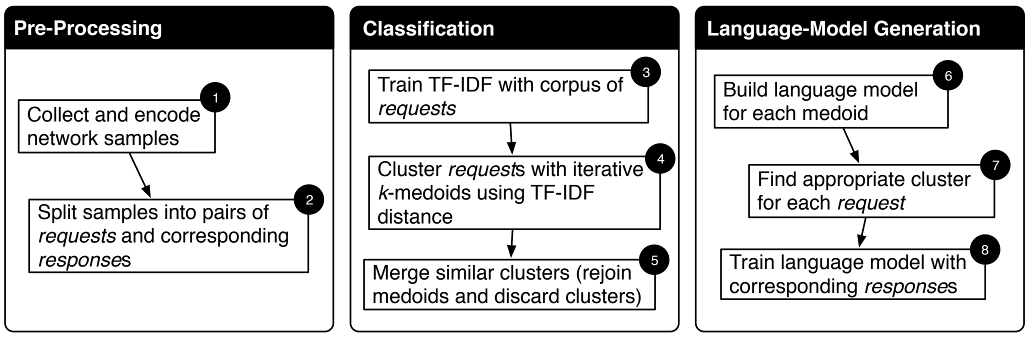

Pre-Processing
Collect and encode network samples
1
Split samples into pairs of *requests* and corresponding *responses*
2
Classification
Train TF-IDF with corpus of *requests*
3
Cluster *requests* with iterative *k*-medoids using TF-IDF distance
4
Merge similar clusters (rejoin medoids and discard clusters)
5
Language-Model Generation
Build language model for each medoid
6
Find appropriate cluster for each *request*
7
Train language model with corresponding *responses*
8

Figure 1: Setup consists of three distinct stages conducted in tandem in preparation for deployment.

classification, and language-model generation. We address each part in turn. We note that although our methodology is not protocol specific, for pedagogical reasons, we provide examples specific to the DNS protocol where appropriate. Our decision to use DNS for validation stems from the fact that validating the correctness of an HTTP response is ill-defined. Likewise, many ASCII-based protocols that come to mind (e.g., HTTP, SMTP, IRC) lack strict notions of correctness and so do not serve as a good conduit to demonstrate the correctness of the output we generate.

To begin, we pre-process and sanitize all trace data used for training. Network traces are stripped of transport protocol headers and organized by session into pairs of requests and responses. Any trace entries that correspond to protocol errors (e.g., HTTP 404) are omitted. Next, we group request and response pairs using a variant of iterative $k$-means clustering with TF/IDF (i.e., term frequency-inverse document frequency) cosine similarity as our distance metric. Formally, we apply a $k$-medoids algorithm for clustering, which assigns samples from the data as cluster medoids (i.e., centroids) rather than numerical averages. For reasons that should become clear later, pair similarity is based solely on the content of the request samples. Upon completion, we then generate and train a collection of smoothed $n$-gram language-models for each cluster. These language-models are subsequently used to produce dynamic responses to online requests. However, because message formats may contain session-specific fields, we also post-process responses to satisfy these dependencies whenever they can be automatically inferred. For example, in DNS, a session identifier uniquely identifies each record request with its response.

During a live deployment, online-classification is used to deduce the response that is most similar to the incoming request (i.e., by mapping the response to its best medoid). For instance, a DNS request for an MX record will ideally match a medoid that maps to other MX requests. The medoid with the minimum TF/IDF distance to an online request identifies which language model is used for generating responses. The language models are built in such a way that they produce responses influenced by the training data. The overall process is depicted in Figure 2. For our evaluation as a web-based honeypot (in Section 6), this process is used in two distinct stages: first, when interacting with search engines for site indexing and second, when courting malware.

## 4 Under the Hood

In what follows, we now present more specifics about our design and implementation. Recall that our goal is to provide a technique for automatically providing valid responses to protocols interactions learned directly from raw traffic.

In lieu of semantic knowledge, we instead apply classic pattern classification techniques for partitioning a set of observed requests. In particular, we use the iterative $k$-medoids algorithm. As our distance metric we choose to forgo byte-sequence alignment approaches that have been previously used to classify similarities between protocol messages (e.g, [18, 9, 6]). As Cui et. al. observed, while these approaches are appropriate for classifying requests that only differ parametrically, byte-sequence alignment is ill-suited for classifying messages with different byte-sequences [8]. Therefore, we use TF/IDF cosine similarity as our distance metric.

Intuitively, *term frequency-inverse document frequency* (TF/IDF) is the measure of a term's significance to a string or document given its significance among a set of documents (or corpus). TF/IDF is often used in information retrieval for a number of applications including automatic text retrieval and approximate string matching [29]. Mathematically, we compute TF/IDF in the following way: let $\tau_{d_i}$ denote how often the term $\tau$ appears in document $d_i$ such that $d_i \in D$, a collection of documents. Then $\texttt{TF/IDF} = \texttt{TF} \cdot \texttt{IDF}$ where

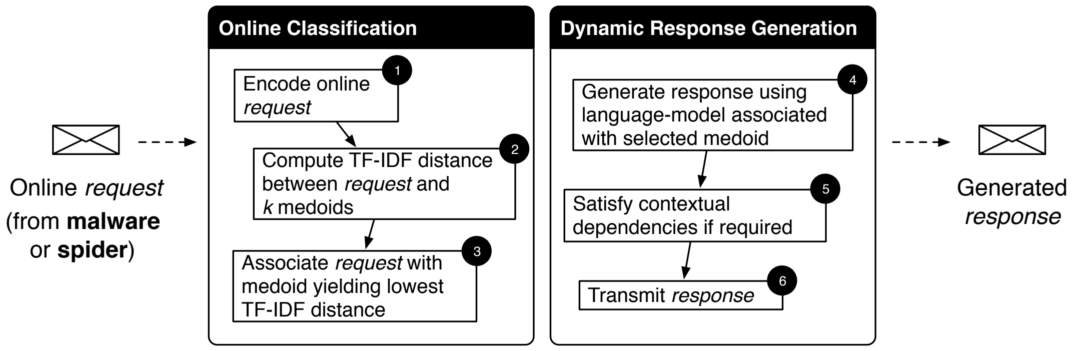

Online Classification
Encode online request
Compute TF-IDF distance between request and k medoids
Associate request with medoid yielding lowest TF-IDF distance
Dynamic Response Generation
Generate response using language-model associated with selected medoid
Satisfy contextual dependencies if required
Transmit response
Online request (from malware or spider)
Generated response
1
2
3
4
5
6

Figure 2: Online classification of requests from malware and search engine spiders influences which language model is selected for response generation.

$$\mathtt{TF} = \sqrt{\tau_{d_i}}$$

and

$$\mathtt{IDF} = \sqrt{\log \frac{|D|}{|\{d_j : d_j \in D \text{ and } \tau \in d_j\}|}}$$

The term-similarity between two strings from the same corpus can be computed by calculating their `TF/IDF` *distance*. To do so, both strings are first represented as multi-dimensional vectors. For each term in a string (e.g., word), its `TF/IDF` value is computed as described previously. Then, for a string with $n$ terms, an $n$-dimensional vector is formed using these values. The cosine of the angle between two such vectors representing strings indicates a measure of their similarity (hence, its complement is a measure of distance).

In the context of our implementation, terms are delineated by tokenizing requests into the following classes: one or more spaces, one or more printable characters (excluding spaces), and one or more non-printable characters (also excluding spaces).[2] We chose the space character as a primary term delimiter due to its common occurrence in text-based protocols; however, the delimiter could have easily been chosen automatically by identifying the most frequent byte in all requests. The collection of all requests (and their constituent terms) form the `TF/IDF` corpus.

Once `TF/IDF` training is complete we use an iterative $k$-*medoids* algorithm, shown in Algorithm 1, to identify similar requests. Upon completion, the classification algorithm produces a $k$ (or less) partitioning over the set of all requests. In an effort to rapidly classify online requests in a memory-efficient manner, we retain only the medoids and dissolve all clusters. For our deployment, we empirically choose $k = 30$, and then perform a trivial cluster-collapsing algorithm: we iterate through the $k$ clusters and, for each cluster, calculate the mean and standard deviation of the distance between the medoid and the other members of the cluster. Once the $k$-means and standard deviations are known, we collapse pairs of clusters if the medoid requests are no more than one standard deviation apart.

## 4.1 Dynamic Response Generation

Since one of the goals of our method is to generate not only valid but also *dynamic* responses to requests, we employ natural language processing techniques (NLP) to create models of protocols. These models, termed *language models*, assign probabilities of occurrence to sequences of tokens based on a corpus of training data. With natural languages such as English we might define a token or, more accurately, a *1-gram* as a string of characters (i.e., a word) delimited by spaces or other punctuation. However, given that we are not working with natural languages, we define a new set of delimiters for protocols. The 1-gram token in our model adheres to one of the following criteria: (1) one or more spaces, (2) one or more printable characters, (3) one or more non-printable characters, or (4) the beginning of message (`BOM`) or end of message (`EOM`) tokens.

The training corpora we use contain both requests and responses. Adhering to our assumption that similar requests have similar responses, we train $k$ response language models on the responses associated with each of the $k$ request clusters. That is, each cluster's response language model is trained on the packets seen in response to the requests in that cluster. Recall that to avoid having to keep every request cluster in memory, we keep only the medoids for each cluster. Then, for each (*request, response*) tuple, we recalculate the distance to each of the $k$ request medoids. The medoid with the minimal distance to the tuple's request identifies which of the $k$ language models is trained using the response. After train-

```
1: $MedoidSet \leftarrow SelectKRandomElements(ObservedRequests)$
2: $RequestMap < RequestType, MedoidType > \leftarrow \perp$ // for mapping requests to medoids
3: repeat
4:   for all $R \in (ObservedRequests - MedoidSet)$ do
5:     for all $M \in MedoidSet$ do
6:       $Distance \leftarrow TF/IDF(R, M)$
7:       if $RequestMap[R] = \perp$ or $Distance < TF/IDF(RequestMap[R], M)$ then
8:         $RequestMap[R] \leftarrow M$
9:   for all $M \in MedoidSet$ do
10:    $M \leftarrow FindMemberWithLowestMeanDistance(M, RequestMap)$
11: until $HasConverged(MedoidSet)$
12: for all $(M_i, M_j) \in MedoidSet$ s.t. $i \neq j$ do
13:   if $TF/IDF(M_i, M_j) < FindThresholdDistance(M_i, M_j)$ then
14:     $MedoidSet \leftarrow MedoidSet - \{M_i, M_j\}$
15:     $MedoidSet \leftarrow MedoidSet \cup Merge(M_i, M_j, RequestMap)$
```

**Algorithm 1:** Iterative $k$-Medoids Classification for Observed Requests

ing concludes, each of the $k$ response language models has a probability of occurrence associated with each observed sequence of 1-grams. A sequence of two 1-grams is called a 2-gram, a sequence of three 1-grams is called a 3-gram, and so on. We cut the maximum $n$-gram length, $n$, to eight.

Since it is unlikely that we have witnessed every possible $n$-gram during training, we use a technique called *smoothing* to lend probability to unobserved sequences. Specifically, we use parametric Witten-Bell back-off smoothing [30], which is the state of the art for $n$-gram models. This smoothing method estimates, if we consider 3-grams, the 3-gram probability by interpolating between the naive count ratio $C(w_1w_2w_3)/C(w_1w_2)$ and a recursively smoothed probability estimate of the 2-gram probability $P(w_3|w_2)$. The recursively smoothed probabilities are less vulnerable to low counts because of the shorter context. A 2-gram is more likely to occur in the training data than a 3-gram and the trend progresses similarly as the $n$-gram length decreases. By smoothing, we get a reasonable estimate of the probability of occurrence for all possible $n$-grams even if we have never seen it during training. Smoothing also mitigates the possibility that certain $n$-grams dominate in small training corpora. It is important to note that during generation, we only consider the states seen in training.

To perform the response generation, we use the language models to define a Markov model. This Markov model can be thought of as a large finite state machine where each transition occurs based on a *transition probability* rather than an input. As well, each "next state" is conditioned solely on the previous state. The transition probability is derived directly from the language models. The transition probability from a 1-gram, $w_1$ to a 2-gram, $w_1w_2$ is $P(w_2|w_1)$, and so on. Intuitively, generation is accomplished by conducting a probabilistic simulation from the start state (i.e., `BOM`) to the end state (i.e., `EOM`).

More specifically, to generate a response, we perform a random walk on the Markov model corresponding to the identified request cluster. From the `BOM` state, we randomly choose among the possible next states with the probabilities present in the language model. For instance, if the letters $(\mathcal{B}, \mathcal{C}, \mathcal{D})$ can follow $\mathcal{A}$ with probabilities $(70\%, 20\%, 10\%)$ respectively, then we will choose the $\mathcal{AB}$ path approximately $70\%$ of the time and similarly for $\mathcal{AC}$ and $\mathcal{AD}$. We use this random walk to create responses similar to those seen in training not only in syntax but also in frequency. Ideally, we would produce the same types of responses with the same frequency as those seen during training, but the probabilities used are at the 1-gram level and not the response packet level.

The Markov models used to generate responses attempt to generate valid responses based on the training data. However, because the training is over the entire set of responses corresponding to a cluster, we cannot recognize contextual dependencies between requests and responses. Protocols will often have session identifiers or tokens that necessarily need to be mirrored between request and response. `DNS`, for instance, has a two byte session identifier in the request that needs to appear in any valid response. As well, the `DNS` name or IP requested also needs to appear in the response. While the NLP engine will recognize that *some* session identifier and domain name should occupy the correct positions in the response, it is unlikely that the *correct* session identifier and name will be chosen. For this reason, we automatically post-process the NLP generated response to appropriately satisfy contextual dependencies.

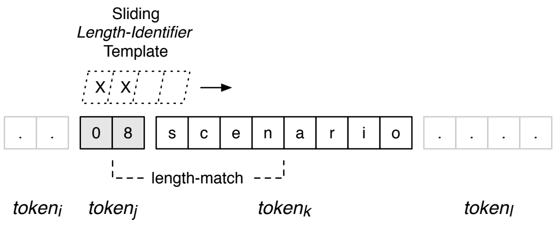


Figure 3: A sliding window template traverses request tokens to identify variable-length tokens that should be reproduced in related responses.

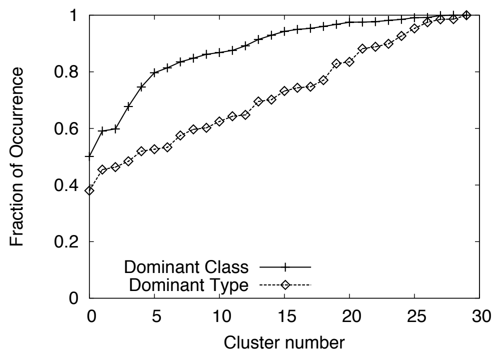


Figure 4: Frequency of dominant type/class per *request* cluster (with $k$ = 30), sorted from least to most accurate.

### 4.1.1 Detecting Contextual Dependencies

Generally speaking, protocols have two classes of contextual dependencies: invariable length tokens and variable length tokens. Invariable length tokens are, as the name implies, tokens that always contain the same number of bytes. For the most part, protocols with variable length tokens typically adhere to one of two standards: tokens preceded by a length field and tokens separated using a special byte delimiter. Overwhelmingly, protocols use length-preceded tokens (`DNS, Samba, Netbios, NFS,` etc.). The other less-common type (as in `HTTP`) employ variable length delimited tokens.

Our method for handling each of these token types differs only slightly from techniques employed by other active responder and protocol disassembly techniques ([8, 9]). Specifically, we identify contextual dependencies using two techniques. First, we apply the Needleman-Wunsch string alignment algorithm [19] to align requests with their associated responses during training. Since the language models we use are not well suited for this particular task, this process is used to identify if, and where, substrings from a request also appear in its response. If certain bytes or sequences of bytes match over an empirically derived threshold (80% in our case), these bytes are considered invariable length tokens and the byte positions are copied from request to response after the NLP generation phase.

To identify variable length tokens, we make the simplifying assumption that these types of tokens are preceded by a length identifier; we do so primarily because we are unaware of any protocols that contain contextual dependencies between request and response through character-delimited variable length tokens. As depicted in Figure 3, we iterate over each request and consider each set of up to four bytes as a length identifier if and only if the token that follows it belongs to a certain character class[3] for the described length. In our example, $Token_i$ is identified as a candidate length-field based upon its value. Since the next immediate token is of the length described by $Token_i$ (i.e., 8), $Token_j$ is identified as a variable length token. For each variable length token discovered, we search for the same token in the observed response. We copy these tokens after NLP generation if and only if this matching behavior was common to more than half of the request and response pairs observed throughout training.

As an aside, the content-length header field in our `HTTP` responses also needs to accurately reflect the number of bytes contained in each response. If the value of this field is greater than the number of bytes in a response, the recipient will poll for more data, causing transactions to stall indefinitely. Similarly, if the value of the content-length field is less than the number of bytes in the response, the recipient will prematurely halt and truncate additional data. While other approaches have been suggested for automatically inferring fields of this type, we simply post-process the generated `HTTP` response and automatically set the content-length value to be the number of bytes after the end-of-header character.

## 5 Validation

In order to assess the correctness of our dynamic response generation techniques, we validate our overall approach in the context of `DNS`. Again, we reiterate that our choice for using `DNS` in this case is because it is a rigid binary protocol, and if we can correctly generate dynamic responses for this protocol, we believe it aptly demonstrates the strength (and soundness) of our approach. For our subsequent evaluation, we train our `DNS` responder off a week's worth of raw network traces collected from a public wireless network used by approximately 50 clients. The traffic was automatically partitioned into request and response tuples as outlined in Section 4.

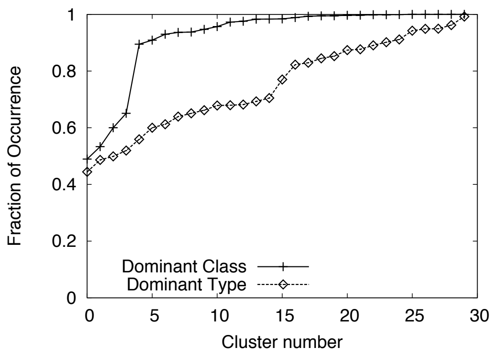


Figure 5: Frequency of dominant type/class per *response* cluster (with $k$ = 30), sorted from least to most accurate.

To validate the output of our clustering technique, we consider clustering of requests successful if for each cluster, one type of request (A, MX, NS, etc.) and the class of the request (IN) emerges as the most dominant member of the cluster; a cluster with one type and one class appearing more frequently than any other is likely to correctly classify an incoming request and, in turn, generate a response to the correct query type and class. We report results based on using 10,000 randomly selected flows for training. As Figures 4 and 5 show, nearly all clusters have a dominating type and class.

To demonstrate our response generation's success rate, we performed 20,000 DNS requests on randomly generated domain names (of varying length). We used the UNIX command host[4] to request several types of records. For validation purposes, we consider a response as strictly faithful if it is correctly interpreted by the requesting program with no warnings or errors. Likewise, we consider a response as valid if it processes correctly with or without warnings or errors. The results are shown in Figure 6 for various training flow sizes. Notice that we achieve a high success rate with as little as 5,000 flows, with correctness ranging between 89% and 92% for strictly faithful responses, and over 98% accuracy in the case of valid responses.

In summary, this demonstrates that the overall design depicted in Figures 1 and 2—that embodies our training phase, classification phase, model generation, and preposing phase to detect contextual dependencies and correctly mirror the representative tokens in their correct location(s)—produces faithful responses. More importantly, these responses are learned automatically, and require little or no manual intervention. In what follows, we further substantiate the utility of our approach in the context of a web-based honeypot.

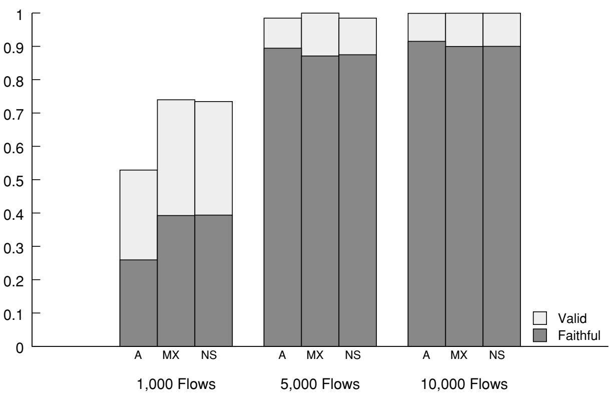


Figure 6: Faithful, valid, or erroneous response success rate for 20,000 random DNS requests under different numbers of training flows.

## 6 Evaluation

Our earlier assertion was that the exploitation of webapps now pose a serious threat to the Internet. In order to gauge the extent to which this is true, we used our dynamic generation techniques to build a lightweight HTTP responder — in the hope of snatching attack traffic targeted at web applications. These attackers query popular search engines for strings that fingerprint the vulnerable software and isolate their targets.

| Type | Appearances |
|---|---|
| .PHP | 3165 |
| .PL | 29 |
| .CGI | 49 |
| .HTML | 15 |
| .PHTML | 2 |

Table 1: Query Types

With this in mind, we obtained a list of the 3,285 of the most searched queries on Google by known botnets attempting to exploit web applications.[5] We then queried Google for the top 20 results associated with each query. Although there are several bot queries that are ambiguous and are most likely not targeting a specific web application, most of the queries were targeted. However, automatically determining the number of different web applications being attacked is infeasible, if not impossible. For this reason, we provide only the break down in the types of web applications being exploited (i.e., PHP, Perl, CGI, etc.) in Table 1.

Nearly all of the bots searching for these queries exploit command injection vulnerabilities. The PHP vulnerabilities are most commonly exploited through remote inclusion of a PHP script, while the Perl vulnerabilities, are usually exploited with UNIX delimiters and commands. Since CGI/HTML/PHTML can house programs from many different types of underlying languages, they encompass a wide range of exploitation techniques. The collected data contains raw traces of the interactions seen when downloading the pages for each of the returned results. Our corpus contained 178,541 TCP flows, of which we randomly selected 24,000 flows as training data for our real-world deployment (see Section 6.1).

Since our primary goal here is to detect (and catch) bots using search engines to query strings present in vulnerable web applications, our responder must be in a position to capture these prey — i.e., it has to be broadly indexed by multiple search engines. To do so, we first created links to our responder from popular pages,[6] and then expedited the indexing process by disclosing the existence of a minor bug in a common UNIX application to the Full-Disclosure mailing list. The bug we disclosed cannot be leveraged for privilege escalation. Bulletins from Full-Disclosure are mirrored on several high-ranking websites and are crawled extensively by search-engine spiders; less than a few hours later, our site appeared in search results on two prominent search engines. And, right on queue, the attacks immediately followed.

## 6.1 Real-World Deployment

For our real-world evaluation, we deployed our system on a 3.0 GHz dual-processor Intel Xeon with 8 GB of RAM. At runtime, memory utilization peaked at 960 MB of RAM when trained with 24,000 flows. CPU utilization remained at negligible levels throughout operation and on average, requests are satisfied in less than a second. Because our design was optimized to purposely keep all data RAM during runtime, disk access was unnecessary.

Shortly after becoming indexed, search-worms began to attack at an alarming rate, with the attacks rapidly increasing over a two month deployment period. During that time, we also recorded the number of indexes returned by Google per day (which totaled just shy of 12,000 during the deployment). We choose to only show PHP attacks because of their prominence. Figure 7 depicts the number of attacks we observed per day. For reference, we provide annotations of our Google index count in ten day intervals until the indices plateau.

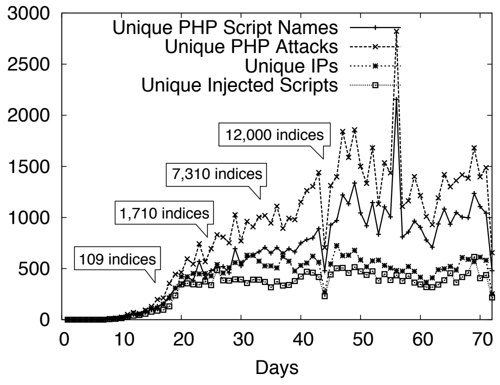


Figure 7: Daily PHP attacks. The valley on day 44 is due to an 8 hr power outage. The peak on day 56 is because two bots launched over 2,000 unique script attacks.

For ease of exposition, we categorize the observed attacks into four groups. The first denotes the number of attacks targeting vulnerabilities that have distinct file structures in their names. The class "Unique PHP attacks", however, is more refined and represents the number of attacks against scripts but using unique injection variables (i.e., `index.php?page=` and `index.php?inc=`). The reason we do so is that the file names and structures can be ubiquitous and so by including the variable names we glean insights into attacks against potentially distinct vulnerabilities. We also attempt to quantify the number of distinct botnets involved in these attacks. While many botnets attack the same application vulnerabilities, (presumably) these botnets can be differentiated by the PHP script(s) they remotely include. Recall that a typical PHP remote-include exploit is of the form "`vulnerable.php?variable=http://site.com/attack\_script?`", and in practice, botnets tend to use disjoint sites to store attack scripts. Therefore, we associate bots with a particular botnet by identifying unique injection script repositories. Based on this admittedly loose notion of uniqueness [27], we observed attacks from 5,648 distinct botnets. Lastly, we record the number of unique IP addresses that attempt to compromise our responder.

The results are shown in Figure 7. An immediate observation is the sheer volume of attacks—in total, well over 368,000 attacks targeting just under 45,000 unique scripts before we shutdown the responder. Interestingly, notice that there are more unique PHP attacks than unique IPs, suggesting that unlike traditional scanning attacks, these bots query for and attack a wide variety of web applications. Moreover, while many bots attempt to exploit a large number of vulnerabilities, the repositories hosting the injected scripts remain unchanged from attack to attack. The range of attacks is perhaps better demonstrated not by the number of unique PHP scripts attacked but by the number of unique PHP web-applications that are the target of these attacks.

#### 6.1.1 Unique WebApps

In general, classifying the number of unique web applications being attacked is difficult because some bots target PHP scripts whose filenames are ubiquitous (e.g., `index.php`). In these cases, bots are either targeting a vulnerability in one specific web-application that happens to use a common filename or arbitrarily attempting to include remote PHP scripts.

To determine if an attack can be linked to a specific web-application, we downloaded the directory structures for over 4,000 web-applications from SourceForge.net. From these directory structures, we matched the web application to the corresponding attacked script (e.g., `gallery.php` might appear only in the Web Gallery web application). Next, we associated an attack with a specific web application if the file name appeared in no more than 10 web-app file structures. We choose a threshold of 10 since SourceForge stores several copies of essentially the same web application under different names (due to, for instance, "skin" changes or different code maintainers). For non-experimental deployments aimed at detecting zero-day attacks, training data could be associated with its application of origin, thereby making associations between non-generic attacks and specific web-applications straightforward.

Based on this heuristic, we are able to map the 24,000 flows we initially trained on to 560 "unique" web-applications. Said another way, by simply building our language models on randomly chosen flows, we were able to generate content that approximates 560 distinct web-applications — a feat that is not as easy to achieve if we were to deploy each application on a typical web-based honeypot (e.g., the Google Hack Honeypot [3]). The attacks themselves were linked back to 295 distinct web applications, which is indicative of the diversity of attacks.

We note that our heuristic to map content to web-apps is strictly a lower bound as it only identifies web-applications that have a distinct directory structure and/or file name; a large percentage of web-applications use `index.php` and other ubiquitous names and are therefore not accounted for. Nonetheless, we believe this serves to make the point that our approach is effective and easily deployable, and moreover, provides insight into the amount of web-application vulnerabilities currently being leveraged by botnets.

#### 6.1.2 Spotting Emergent Threats

While the original intention of our deployment was to elicit interaction from malware exploiting known vulnerabilities in web applications, we became indexed under broader conditions due to the high amount of variability in our training data. As a result, a honeypot or active responder indexed under such a broad set of web applications can, in fact, attract attacks targeting *unknown* vulnerabilities. For instance, according to `milw0rm` (a popular security advisory/exploit distribution site), over 65 `PHP` remote inclusion vulnerabilities were released during our two month deployment [1]. Our deployment began on October $27^{th}$, 2007 and used the *same* training data for its entire duration. Hence, any attack exploiting a vulnerability released after October $27^{th}$ is an attack we did not explicitly set out to detect.

Nonetheless, we witnessed several emergent threats (some may even consider them "zero-day" attacks) because some of the original queries used to bootstrap training were generic and happened to represent a wide number of webapps. As of this writing, we have identified more than 10 attacks against vulnerabilities that were undisclosed at deployment time (some examples are illustrated in Table 2). It is unlikely that we witnessed these attacks simply because of arbitrary attempts to exploit random websites—indeed, we never witnessed many of the other disclosed vulnerabilities being attacked.

We argue that given the frequency with which these types of vulnerabilities are released, a honeypot or an active responder without dynamic content generation will likely miss an overwhelming amount of attack traffic—in the attacks we witnessed, botnets begin attacking vulnerable applications on the day the vulnerability was publicly disclosed! An even more compelling case for our architecture is embodied by attacks against vulnerabilities that have not been disclosed (e.g., the recent WordPress vulnerability [7]). We believe that the potential to identify these attacks exemplifies the real promise of our approach.

## 6.2 Dissecting the Captured Payloads

To better understand what the post-infection process entails, we conducted a rudimentary analysis of the remotely included `PHP` scripts. Our malware analysis was performed on a Linux based Intel virtual machine with the 2.4.7 kernel. We used a deprecated kernel version since newer versions do not export the system call table of which we take advantage. Our environment consisted of a kernel module and a preloaded library[7] that serve to inoculate malware before execution and to log interesting behavior. The preloaded library captures calls

| Disclosure Date | Attack Date | Signature |
|---|---|---|
| 2007-11-04 | 2007-11-10 | `/starnet/themes/c-sky/main.inc.php?cmsdir=` |
| 2007-11-21 | 2007-11-23 | `/comments-display-tpl.php?language_file=` |
| 2007-11-22 | 2007-11-22 | `/admin/kfm/initialise.php?kfm_base_path=` |
| 2007-11-25 | 2007-11-25 | `/Commence/includes/db_connect.php?phproot\_path=` |
| 2007-11-28 | 2007-11-28 | `/decoder/gallery.php?ccms_library_path=` |

Table 2: Attacks targeting vulnerabilities that were unknown at time of deployment

to `connect()` and `send()`. The `connect` hook deceives the malware by faking successful connections, and the `send` function allows us to record information transmitted over sockets.[8]

Our kernel module hooks three system calls: (`open`, `write`, and `execve`). We execute every script under a predefined user ID, and interactions under this ID are recorded via the `open()` hook. We also disallow calls to `open` that request write access to a file, but feign success by returning a special file descriptor. Attempts to write to this file descriptor are logged via syslog. Doing so allows us to record files written by the malware without allowing it to actually modify the file system. Similarly, only commands whose file names contain a pre-defined random password are allowed to execute. All other command executions under the user ID fail to execute (but pretend to succeed), assuring no malicious commands execute. Returning success from failed executions is important because a script may, for example, check if a command (e.g., `wget`) successfully executes before requesting the target URL.

To determine the functionality of the individual malware scripts, we batched processed all the captured malware on the aforementioned architecture. From the transcripts provided by the kernel module and library, we were able to discern basic functionality, such as whether or not the script makes connections, issues IRC commands, attempts to write files, etc. In certain cases, we also conducted more in-depth analyses by hand to uncover seemingly more complex functionality. We discuss our findings in more detail below.

The high-level break-down for the observed scripts is given in Table 3. The challenge in capturing bot payloads in web application attacks stems from the ease with which the attacker can test for a vulnerability; unique string displays (where the malware echoes a unique token in the response to signify successful exploitation) accounts for the most prevalent type of injection. Typically, bots parse returned responses for their identifying token and, if found, proceed to inject the actual bot payload. Since these unique tokens are unlikely to appear in our generated response, we augment our responder to echo these tokens at run-time. While the use of random numbers as tokens seem to be the *soup du jour* for testing

| Script Classification | Instances |
|---|---|
| PHP Web-based Shells | 834 |
| Echo Notification | 591 |
| PHP Bots | 377 |
| Spammers | 347 |
| Downloaders | 182 |
| Perl Bots | 136 |
| Email Notification | 87 |
| Text Injection | 35 |
| Java-script Injection | 18 |
| Information Farming | 9 |
| Uploaders | 4 |
| Image Injection | 4 |
| UDP Flooders | 3 |

Table 3: Observed instances of individual malware

a vulnerability, we observed several instances where attackers injected an image. Somewhat comically, in many cases, the bot simply e-mails the IP address of the vulnerable machine, which the attacker then attempts to exploit at a later time. The least common vulnerability test we observed used a connect-back operation to connect to an attacker-controlled system and send vulnerability information to the attacker. This information is presumably logged server-side for later use.

Interestingly, we notice that bots will often inject simple text files that typically also contain a unique identifying string. Because PHP scripts can be embedded inside HTML, PHP requires begin and end markers. When a text file is injected without these markers, its contents are simply interpreted as HTML and displayed in the output. This by itself is not particularly interesting, but we observed several attackers injecting large lists of queries to find vulnerable web applications via search engines. The largest query list we captured contained 7,890 search queries that appear to identify vulnerable web applications — all of which could be used to bootstrap our content generation further and cast an even wider net.

Overall, the collected malware was surprisingly modular and offered diverse functionality similar to that reported elsewhere [26, 15, 13, 12, 25, 5, 4]. The captured scripts (mostly PHP-based command shells), are advanced enough that many have the ability to display the output in some user-friendly graphical user interface, obfuscate the script itself, clean the logs, erase the script and related evidence, deface a site, crawl vulnerability sites, perform distributed denial of service attacks and even perform automatic self-updates. In some cases, the malware inserted tracking cookies and/or attempted to gain more information about a system's inner-workings (e.g., by copying `/etc/passwd` and performing local banner scans). To our surprise, only eight scripts contained functionality to automatically obtain `root`. In these cases, they all used C-based kernel vulnerabilities that write to the disk and compile upon exploitation. Lastly, IRC was used almost exclusively as the communication medium. As can be expected, we also observed several instances of spamming malware using e-mail addresses pulled from the web-application's `MySQL` database backend. In a system like `phpBB`, this can be highly effective because most forum users enter an e-mail address during the registration process. Cross-checking the bot IPs with data from the Spamhaus project [2] shows that roughly 36% of them currently appear in the spam black list.

One noteworthy functionality that seems to transcend our categorizations among `PHP` scripts is the ability to break out of `PHP` safe mode. `PHP` safe mode disables functionality for, among others, executing system commands, modifying the file system, etc. The malware we observed that bypass safe mode tend to contain a handful of known exploits that either exploit functionality in `PHP`, functionality in mysql, or functionality in web server software. Lastly, we note that although we observed what appeared to be over 5,648 unique injection scripts from distinct botnets, nearly half of them point to zombie botnets. These botnets no longer have a centralized control mechanism and the remotely included scripts are no longer accessible. However, they are still responsible for an overwhelming amount of our observed `HTTP` traffic.

## 6.3 Limitations

One might argue that a considerably less complex (but more mundane) approach for eliciting search worm traffic may be to generate large static pages that contain content representative of a variety of popular web-applications. However, simply returning arbitrary or static pages does not yield either the volume or diversity of attacks we observed. For instance, one of our departmental websites (with a much higher PageRank than our deployment site) only witnessed 437 similar attacks since August 2006. As we showed in Section 6, we witnessed well over 368,000 attacks in just over two months. Moreover, close inspection of the attacks on the university website show that they are far less varied or interesting. These attacks seem to originate from either a few botnets that issue "loose" search queries (e.g., "inurl:index.php") and subsequently inject their attack, or simply attack ubiquitous file names with common variable names. Not surprisingly, these unsophisticated botnets are less widespread, most likely because they fail to infect many hosts. By contrast, the success of our approach lead to more insightful observations about the scope and diversity of attacks because we were able to cast a far wider net.

That said, for real-world honeypot deployments, detection and exploitation of the honeypot itself can be a concern. Clearly, our system is not a true web-server and like other honeypots [23], it too can be trivially detected using various fingerprinting techniques [14]. More to the point, a well-crafted bot that knows that a particular string always appears in pages returned by a given web-application could simply request the page from us and check for the presence of that string. Since we will likely fail to produce that string, our phony will be detected[9].

The fact that our web-honeypot can be detected is a clear limitation of our approach, but in practice it has not hindered our efforts to characterize current attack trends, for several reasons. First, the search worms we witnessed all seemed to use search engines to find the identifying information of a web-application, and attacked the vulnerability upon the first visit to the site; presumably because verifying that the response contains the expected string slows down infection. Moreover, it is often times difficult to discern the web-application of origin as many web-applications do not necessarily contain strings that uniquely identify the software. Indeed, in our own analysis, we often had difficulty identifying the targeted web-application by hand, and so automating this might not be trivial.

Lastly, we argue that the limitations of the approach proposed herein manifests themselves as trade-offs. Our decision to design a stateless system results in a memory-efficient and lightweight deployment. However, this design choice also makes handling stateful protocols nearly impossible. It is conceivable that one can convert our architecture to better interact with stateful protocols by simply changing some aspects of the design. For instance, this could be accomplished by incorporating flow sequence information into training and then recalling its hierarchy during generation (e.g., by generating a response from the set of appropriate first round responses, then second round responses, etc.). To capture multi-stage attacks, however, *ScriptGen* [18, 17] may be a better choice for emulating multi-stage protocol interaction, and can be used in conjunction with our technique to cast a wider net to initially entice such malware.

## 7 Conclusion

In this paper, we use a number of multi-disciplinary techniques to generate dynamic responses to protocol interactions. We demonstrate the utility of our approach through the deployment of a dynamic content generation system targeted at eliciting attacks against web-based exploits. During a two month period we witnessed an unrelenting barrage of attacks from attackers that scour search engine results to find victims (in this case, vulnerable web applications). The attacks were targeted at a diverse set of web applications, and employed a myriad of injection techniques. We believe that the results herein provide valuable insights on the nature and scope of this increasing Internet threat.

## 8 Acknowledgments

We thank Ryan MacArthur for his assistance with the forensic analysis presented in Section 6.2. We are grateful to him for the time spent analyzing and cataloging the captured payloads. We also thank David Dagon for cross-referencing bot IP addresses with the Spamhaus Project black lists [2]. We extend our gratitude to our shepherd, George Danezis, and the anonymous reviewers for their invaluable comments and suggestions. We also thank Bryan Hoffman, Charles Wright, Greg MacManus, Moheeb Abu Rajab, Lucas Ballard, and Scott Coull for many insightful discussions during the course of this project. This work was funded in part by NFS grants CT-0627476 and CNS-0546350.

## Data Availability

To promote further research and awareness of the malware problem, the data gathered during our live deployment is available to the research community. For information on how to get access to this data, please see `http:/spar.isi.jhu.edu/botnet_data/`.

## References

[1] Milw0rm. See `http://www.milw0rm.com/`.

[2] The Spamhaus Project. See `http://www.spamhaus.org/`.

[3] The Google Hack Honeypot, 2005. See `http://ghh.sourceforge.net/`.

[4] ANDERSON, D. S., FLEIZACH, C., SAVAGE, S., AND VOELKER, G. M. Spamscatter: Characterizing internet scam hosting infrastructure. In *Proceedings of the 16th USENIX Security Symposium*, pp. 135–148.

[5] BARFORD, P., AND YEGNESWARAN, V. An inside look at botnets. In *Advances in Information Security* (2007), vol. 27, Springer Verlag, pp. 171–191.

[6] BEDDOE, M. The protocol informatics project, 2004.

[7] CHEUNG, A. Secunia's wordpress GBK/Big5 character set "S" SQL injection advisory. See `http://secunia.com/advisories/28005/`.

[8] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium* (Boston, MA, August 2007), pp. 199–212.

[9] CUI, W., PAXSON, V., WEAVER, N., AND KATZ, R. H. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium 2006* (February 2006), Internet Society.

[10] CUI, W., PAXSON, V., AND WEAVER, N. C. GQ: Realizing a system to catch worms in a quarter million places. Tech. Rep. TR-06-004, International Computer Science Institute, 2006.

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 211–224.

[12] FREILING, F. C., HOLZ, T., AND WICHERSKI, G. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)* (September 2005), vol. 3679 of *Lecture Notes in Computer Science*, pp. 319–335.

[13] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium* (August 2007), pp. 167–182.

[14] HOLZ, T., AND RAYNAL, F. Detecting honeypots and other suspicious environments. In *Proceedings of the Workshop on Information Assurance and Security* (June 2005).

[15] Know your enemy: Tracking botnets. Tech. rep., The Honeynet Project and Research Alliance, March 2005. Available from `http://www.honeynet.org/papers/bots/`.

[16] KREIBICH, C., AND CROWCROFT, J. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)* (Boston, November 2003).

[17] LEITA, C., DACIER, M., AND MASSICOTTE, F. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *RAID* (2006), D. Zamboni and C. Krügel, Eds., vol. 4219 of *Lecture Notes in Computer Science*, Springer, pp. 185–205.

[18] LEITA, C., MERMOUD, K., AND DACIER, M. ScriptGen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference* (December 2005), pp. 203–214.

[19] NEEDLEMAN, S. B., AND WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48 (1970), 443–453.

[20] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 226–241.

[21] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium* (2005).

[22] PANG, R., YEGNESWARAN, V., BARFORD, P., PAXSON, V., AND PETERSON, L. Characteristics of Internet background radiation, October 2004.

[23] PROVOS, N. A virtual honeypot framework. In *Proceedings of the 12th USENIX Security Symposium* (August 2004), pp. 1–14.

[24] PROVOS, N., MCCLAIN, J., AND WANG, K. Search worms. In *Proceedings of the 4th ACM workshop on Recurring malcode* (New York, NY, USA, 2006), ACM, pp. 1–8.

[25] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser: Analysis of web-based malware. In *Usenix Workshop on Hot Topics in Botnets (HotBots)* (2007).

[26] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference* (October 2006), ACM, pp. 41–52.

[27] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging. In *Proceedings of the first USENIX workshop on hot topics in Botnets (HotBots '07).* (April 2007).

[28] RIDEN, J., MCGEEHAN, R., ENGERT, B., AND MUETER, M. Know your enemy: Web application threats, February 2007.

[29] TATA, S., AND PATEL, J. Estimating the selectivity of TF-IDF based cosine similarity predicates. *SIGMOD Record 36*, 2 (June 2007).

[30] WITTEN, I. H., AND BELL, T. C. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory 37*, 4 (1991), 1085–1094.

[31] YEGNESWARAN, V., GIFFIN, J. T., BARFORD, P., AND JHA, S. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD, USA, Aug. 2005), pp. 97–112.

## Notes

[1] The drawback, of course, is that high-interaction honeypots are a heavy-weight solution, and risk creating their own security problems [23].

[2] Protocol messages are tokenized similarly in [18, 17] and [8].

[3] In practice, we use printable and non-printable.

[4] The results are virtually the same for `nslookup`, and hence, omitted.

[5] These initial queries were provided by one of the authors, but similar results could easily be achieved by crawling the WebApp directories in SourceForge and searching Google for identifiable strings (similar to what we outline in Section 6.1.1).

[6] We placed links on 3 pages with Google PageRank ranking of 6, 2 pages with rank 5, 3 pages with rank 2, and 5 pages with rank 0.

[7] A preloaded library loads before all other libraries in order to hook certain library functions

[8] Because none of the malware we obtained use direct system calls to either `connect()` or `send()`, this setup suffices for our needs.

[9] Notice however that if a botnet has $n$ bots conducting an attack against a particular web-application, we only need to probabilistically return what the malware is seeking $1/n^{th}$ of the time to capture the malicious payload.

# Reverse-Engineering a Cryptographic RFID Tag

Karsten Nohl and David Evans
*University of Virginia*
*Department of Computer Science*
{nohl,evans}@cs.virginia.edu

Starbug and Henryk Plötz
*Chaos Computer Club*
*Berlin*
starbug@ccc.de, henryk@ploetzli.ch

**Abstract**

The security of embedded devices often relies on the secrecy of proprietary cryptographic algorithms. These algorithms and their weaknesses are frequently disclosed through reverse-engineering software, but it is commonly thought to be too expensive to reconstruct designs from a hardware implementation alone. This paper challenges that belief by presenting an approach to reverse-engineering a cipher from a silicon implementation. Using this mostly automated approach, we reveal a cipher from an RFID tag that is not known to have a software or micro-code implementation. We reconstruct the cipher from the widely used Mifare Classic RFID tag by using a combination of image analysis of circuits and protocol analysis. Our analysis reveals that the security of the tag is even below the level that its 48-bit key length suggests due to a number of design flaws. Weak random numbers and a weakness in the authentication protocol allow for pre-computed rainbow tables to be used to find any key in a matter of seconds. Our approach of deducing functionality from circuit images is mostly automated, hence it is also feasible for large chips. The assumption that algorithms can be kept secret should therefore to be avoided for any type of silicon chip.

*Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi.*
([A cipher] must not depend on secrecy, and it must not matter if it falls into enemy hands.)
August Kerckhoffs, *La Cryptographie Militaire*, January 1883 [13]

## 1 Introduction

It has long been recognized that security-through-obscurity does not work. However, vendors continue to believe that if an encryption algorithm is released only as a hardware implementation, then reverse-engineering the cipher from hardware alone is beyond the capabilities of likely adversaries with limited funding and time. The design of the cipher analyzed in this paper, for example, had not been disclosed for 14 years despite more than a billion shipped units. We demonstrate that the cost of reverse engineering a cipher from a silicon implementation is far lower than previously thought.

In some cases, details of an unknown cryptographic cipher may be found by analyzing the inputs and outputs of a black-box implementation. Notable examples include Bletchley Park's breaking the Lorenz cipher during World War II without ever acquiring a cipher machine [5] and the disclosure of the DST cipher used in cryptographic Radio Frequency Identification (RFID) tokens from Texas Instruments [4]. In both cases, researchers started with a rough understanding of the cipher's structure and were able to fill in the missing details through cryptanalysis of the cipher output for known keys and inputs. This black-box approach requires some prior understanding of the structure of a cipher and is only applicable to ciphers with statistical weaknesses. The output of a sound cipher should not be statistically biased and therefore should not leak information about its structure.

Other ciphers have been disclosed through disassembly of their software implementation. Such implementations can either be found in computer software or as microcode on an embedded micro-controller. Ciphers found through software disassembly include the A5/1 and A5/2 algorithms that secure GSM cell phone communication [1] and the Hitag2 and Keeloq algorithms used in car remote controls [3]. The cryptography on the RFID tags we analyzed is not known to be available in software or in a micro-code implementation; tags and reader chips implement the cipher entirely in hardware.

In this paper, we focus on revealing proprietary cryptography from its silicon implementation alone. Reverse-engineering silicon is possible even when very little is known about a cipher and no software implementation exists. The idea of reverse-engineering hardware is not new. Hardware analysis is frequently applied in industry, government, and the military for spying, security assessments, and protection of intellectual property. Such reverse-engineering, however, is usually considered prohibitively expensive for typical attackers, because of the high prices charged by professionals offering this service. The key contribution of this work is demonstrating that reverse-engineering silicon is cheap and that it can be mostly automated. This is the first published work to describe the details of reverse-engineering a cryptographic function from its silicon implementation. We describe a mostly automated process that can be used to cheaply determine the functionality of previously unknown cipher implementations.

We demonstrate the feasibility of our approach by revealing the cipher implemented on the NXP Mifare Classic RFID tags, the world's most widely used cryptographic RFID tag [16]. Section 2 describes our reverse-engineering method and presents the cipher. Section 3 discusses several weaknesses in the cipher beyond its short key size. Weak random numbers combined with a protocol flaw allow for rainbow tables to be computed that reduce the attack time from weeks to minutes. Section 4 discusses some potential improvements and defenses. While we identify fixes that would increase the security of the Mifare cipher significantly, we conclude that good security may be hard to achieve within the desired resource constraints.

## 2 Mifare Crypto-1 Cipher

We analyzed the Mifare Classic RFID tag by NXP (formerly Philips). This tag has been on the market for over a decade with over a billion units sold. The Mifare Classic card is frequently found in access control systems and tickets for public transport. Large deployments include the Oyster card in London, and the SmartRider card in Australia. Before this work, the Netherlands were planning to deploy Mifare tags in OV-chipkaart, a nationwide ticketing system, but the system will likely be re-engineered after first news about a potential disclosure of the card's details surfaced [17]. The Mifare Classic chip currently sells for 0.5 Euro in small quantities, while tags with larger keys and established ciphers such as 3-DES are at least twice as expensive.

The cryptography found in the Mifare cards is a stream cipher with 48-bit symmetric keys. This key length has been considered insecure for some time (for example, the Electronic Frontier Foundation's DES cracking machine demonstrated back in 1998 that a moderately-funded attacker could brute force 56-bit DES [6]) and the practical security that Mifare cards have experienced in the past relies primarily on the belief that its cipher was secret. We find that the security of the Mifare Classic is even weaker than the short key length suggests due to flaws in its random number generation and the initialization protocol discussed in Section 3.

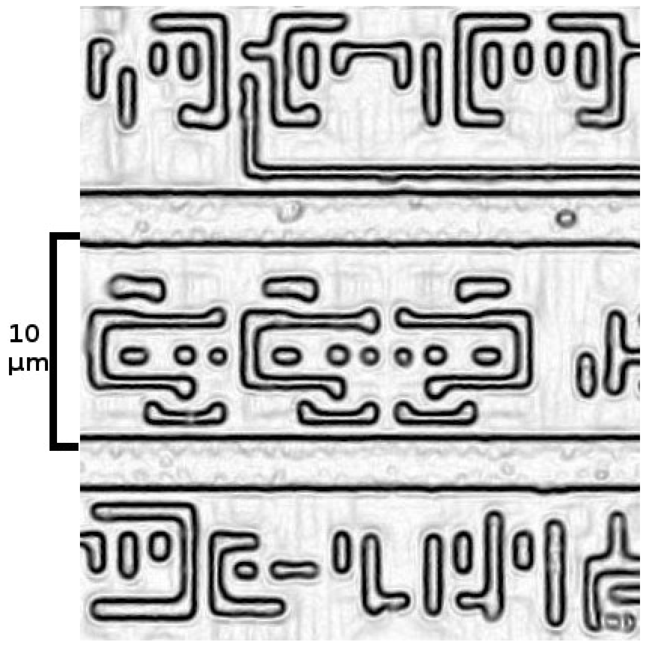
The data on the Mifare cards is divided into sectors, each of which holds two different keys that may have different access rights (e.g., read/write or read-only). This division allows for different applications to each store encrypted data on a tag—an option rarely used in practice. All secrets are set to default values at manufacturing time but changed before issuing the tags to users. Different tags in a system may share the same read key or have different keys. Sharing read keys minimizes the overhead of key-distribution to offline readers. We find, however, that the protocol level measures meant to prevent different users from impersonating each other are insufficient. Unique read and write keys should, therefore, be used for each tag and offline readers should be avoided as much as possible.

### 2.1 Hardware Analysis

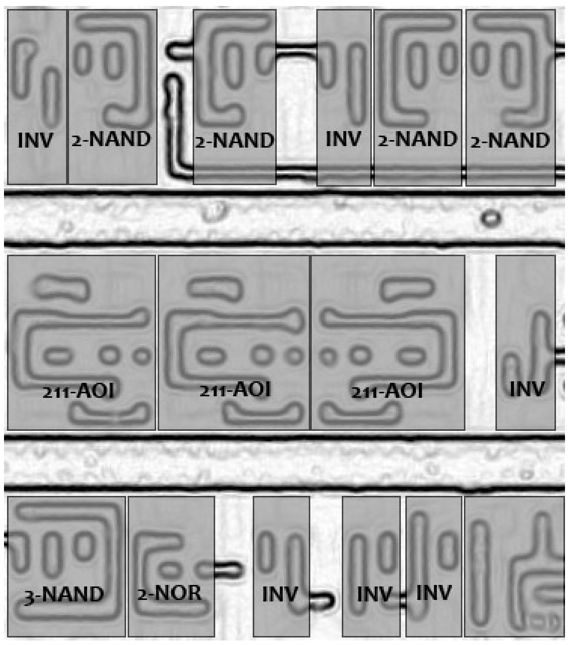
The chip on the Mifare Classic tag is very small with a total area of roughly one square millimeter. About a quarter of the area is used for 1K of flash memory (a 4K version is also available); another quarter is occupied by the radio front-end and outside connectivity, leaving about half the chip area for digital logic including cryptography.

The cryptography functions make up about 400 2-NAND gate equivalents (GE), which is very small even compared to highly optimized implementations of standard cryptography. For example, the smallest known implementation of the AES block cipher (which was specifically designed for RFID tags) requires 3400 GEs [7]. The cryptography on the Mifare tags is also very fast and outputs 1 bit of key stream in every clock cycle. The AES circuit, by comparison, takes 1000 clock cycles for one 128-bit AES operation (10 milliseconds on a tag running at 106 kHz).

To reverse engineer the cryptography, we first had to get access to sample chips, which are usually embedded in credit card size plastic cards. We used acetone to dissolve the plastic card, leaving only the blank chips. Acetone is easier and safer to handle than alternatives such as fuming nitric acid, but still dissolves plastic cards in



(a)



(b)

Figure 1: (a) Source image of layer 2 after edge detection; (b) after automated template detection.

about half an hour. Once we had isolated the silicon chips, we removed each successive layer through mechanical polishing, which we found easier to control than chemical etching. Simple polishing emulsion or sandpaper with very fine grading of $0.04\mu m$ suffices to take off micrometer-thick layers within minutes.

Although the polishing is mostly straightforward, the one obstacle to overcome is the chip tilting. Since the chip layers are very close together, even the smallest tilt leads to cuts through several layers. We addressed this problem in two ways. First, we embedded the millimeter-size chip in a block of plastic so it was easier to handle. Second, we accpeted that we could not completely avoid tilt using our simple equipment and adapted our image stitching tools to patch together chip layers from several sets of pictures, each imaging parts of several layers.

The chip contains a total of six layers, the lowest of which holds the transistors. We took pictures using a standard optical microscope at a magnification of 500x. From multiple sets of these images we were able to automatically generate images of each layer using techniques for image tiling that we borrowed from panorama photography. We achieved the best results using the open source tool `hugin` (http://hugin.sourceforge.net/) by setting the maximum variance in viewer angle to a very small value (e.g., 0.1°) and manually setting a few control points on each image.

The transistors are grouped in gates that each perform a logic function such as AND, XOR, or flip-flop as illustrated in Figure 1. Across the chip there are several thousand such logic gates, but only about 70 different types of gates. As a first step toward reconstructing the circuit, we built a library of these gates. We implemented template matching that given one instance of a logic gate finds all the other instances of the same gate across the chip. Our tools take as input an image of layer 2, which represents the logic level, and the position of instances of different logic gates in the image. The tools then use template matching to find all other instances of the gate across the image, including rotated and mirrored variants. Since larger gates sometimes contain smaller gates as building blocks, the matching is done in order of decreasing gate sizes.

Our template matching is based on *normalized cross-correlation* which is a well-known similarity test [14] and implemented using the MATLAB image processing library. Computing this metric is computationally more complex than standard cross-correlation, but the total running time of our template matching is still under ten minutes for the whole chip. Normalized cross-correlation is insensitive to the varying brightness across our different images and the template matching is able to find matches with high accuracy despite varying coloration and distortion of the structures that were caused by the polishing.

We then manually annotated each type of gate with its respective functionality. This step could be automated as well through converting the silicon-level depiction of each gate into a format suitable for a circuit simulation program. We decided against this approach because the overhead seemed excessive. For larger libraries that perhaps intentionally vary the library cells in an attempt to impede reverse-engineering, however, automation is certainly possible and has already been demonstrated in other projects [2].

Our template matching provides a map of the different logic gates across the chip. While it would certainly have been possible to reverse-engineer the whole RFID tag, we focused our attention on finding and reconstructing the cryptographic components. We knew that the stream cipher would have to include at least a 48-bit register and a number of XOR gates. We found these components in one of the corners of the chip along with a circuit that appeared to be a random number generator as it has an output, but no input.

Focusing our efforts on only these two parts of the chip, we reconstructed the connections between all the logic gates. This step involved considerable manual effort and was fairly error-prone. All the errors we made were found through a combination of redundant checking and statistical tests for some properties that we expected the cipher to have such as an even output distribution of blocks in the filter function. We have since implemented scripts to automate the detection of wires, which can speed the process and improve its accuracy. Using our manually found connections as ground truth we find that our automated scripts detect the metal connection and intra-layer vias correctly with reasonably high probability. In our current tests, our scripts detect over 95% of the metal connections correctly and the few errors they make were easily spotted manually by overlaying the source image and the detection result. These results are, however, preliminary, as many factors are not yet accounted for. To assess the potential for automation more thoroughly, we plan to test our tools on different chips, using different imaging systems, and having different users check the results.

In the process of reconstructing the circuit, we did not encounter any added obscurity or tamper-proofing. Because the cryptographic components are highly structured, they were particularly easy to reconstruct. Furthermore, we could test the validity of different building blocks by checking certain statistical properties. For example, the different parts of the filter function each have an even output distribution so that the output bits are not directly disclosing information about single state bits.

The map of logic gates and the connections between them provides us with almost enough information to discover the cryptographic algorithm. Because we did not reverse-engineer the control logic, we do not know the exact timing and inputs to the cipher. Instead of reconstructing more circuitry, we derived these missing pieces of information from protocol layer communication between the Mifare card and reader.

## 2.2 Protocol Analysis

From the discovered hardware circuit, we could not derive which inputs are shifted into the cipher in what order, partly because we did not reverse the control logic, but also because even with complete knowledge of the hardware we would not yet have known what data different memory cells contain. To add the missing details to the cipher under consideration, and to verify the results of the hardware analysis, we examined communication between the Mifare tags and a Mifare reader chip.

An NXP reader chip is included on the OpenPCD open source RFID reader, whose flexibility proved to be crucial for the success of our project. The OpenPCD includes an ARM micro-controller that controls the communication between the NXP chip and the Mifare card. This setup allows us to record the communication and provides full control over the timing of the protocol. Through timing control we can amplify some of the vulnerabilities we discovered as discussed in Section 3.

No details of the cipher have been published by the manufacturer or had otherwise been leaked to the public prior to this work. We guessed that the secret key and the tag ID were shifted into the shift register sequentially rather than being combined in a more complicated way. To test this hypothesis, we checked whether a reader could successfully authenticate against a tag using an altered key and an altered ID. Starting with single bit changes in ID and key and progressively extending our search to larger variations, we found a number of such combinations that indeed successfully authenticated the reader to the tag. From the pattern of these combinations we could derive not just the order of inputs, but also the structure of the linear feedback shift register, which we had independently found on the circuit level. Combining these insights into the authentication protocol with the results of our hardware analysis gave us the whole Crypto-1 stream cipher, shown in Figure 2.
The cipher is a single 48-bit linear feedback shift register (LFSR). From a fixed set of 20 state bits, the one bit of key stream is computed in every clock cycle. The shift register has 18 taps (shown as four downward arrows in

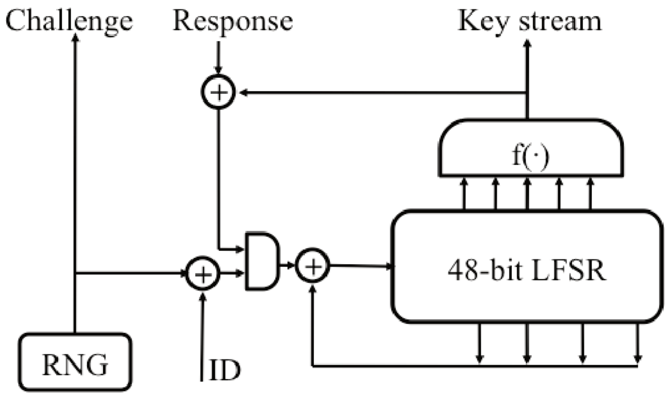


Figure 2: Crypto-1 stream cipher and initialization.

the figure) that are linearly combined to fill the first register bit on each shift. The update function does not contain any non-linearity, which by today's understanding of cipher design can be considered a serious weakness. The generating polynomial of the register is (with $x^i$ being the $i$th bit of the shift register):

$$\begin{aligned} & x^{48}+x^{43}+x^{39}+x^{38}+x^{36}+x^{34}+x^{33}+x^{31}+x^{29} \\ + \quad & x^{24}+x^{23}+x^{21}+x^{19}+x^{13}+x^{9}+x^{7}+x^{6}+x^{5}+1. \end{aligned}$$

The polynomial is *primitive* in the sense that it is irreducible and generates all $\left(2^{48}-1\right)$ possible outputs in succession. To confirm this, we converted the Fibonacci LFSR into a Galois LFSR for which we can compute any number of steps in a few Galois field multiplications. We then found that the cipher state repeats after $\left(2^{48}-1\right)$ steps, but not after any of the possible factors for this number. The LSFR is hence of maximum-length.

The protocol between the Mifare chip and reader loosely follows the ISO 9798-2 specification, which describes an abstract challenge-response protocol for mutual authentication. The authentication protocol takes a shared secret key and a unique tag ID as its inputs. At the end of the authentication, the parties have established a session key for the stream cipher and both parties are convinced that the other party knows the secret key.

## 3 Cipher Vulnerabilities

The 48-bit key used in Mifare cards makes brute-force key searches feasible. Cheaper than brute-force attacks, however, are possible because of the cipher's weak cryptographic structure. While the vulnerability to brute-force attacks already makes the cipher weak, the cheaper attacks are relevant for many Mifare deployments such as fare collection where the value of breaking a particular key is relatively low. Weaknesses of the random number generator and the cryptographic protocol allow an attacker to pre-compute a codebook and perform key-lookups quickly and cheaply using rainbow tables.

### 3.1 Brute-Force Attack

In a brute-force attack an attacker records two challenge-response exchanges between the legitimate reader and a card and then tries all possible keys for whether they produce the same result.

To estimate the expected time for a brute-force attack, we implemented the cipher on FPGA devices by Pico Computing. Due to the simplicity of the cipher, 6 fully-pipelined instances can be squeezed into a single Xilinx Virtex-5 LX50 FPGA. Running the implementation on an array of 64 such FPGAs to try all $2^{48}$ keys takes under 50 minutes.

### 3.2 Random Number Generation

The random number generator (RNG) used on the Mifare Classic tags is highly insecure for cryptographic applications and further decreases the attack complexity by allowing an attacker to pre-compute a codebook.
The random numbers on Mifare Classic tags are generated using a linear feedback shift register with constant initial condition. Each random value, therefore, only depends on the number of clock cycles elapsed between the time the tag is powered up (and the register starts shifting) and the time the random number is extracted. The numbers are generated using a maximum length 16-bit LFSR of the form:

$$x^{16} + x^{14} + x^{13} + x^{11} + 1.$$

The register is clocked at 106 kHz and wraps around every 0.6 seconds after generating all 65,535 possible output values. Aside from the highly insufficient length of the random numbers, an attacker that controls the timing of the protocol controls the generated number. The weakness of the RNG is amplified by the fact that the generating LFSR is reset to a known state every time the tag starts operating. This reset is completely unnecessary, involves hardware overhead, and destroys the randomness that previous transactions and unpredictable noise left in the register.

We were able to control the number the Mifare random number circuit generated using the OpenPCD reader and custom-built firmware. In particular, we were able to generate the same "random" nonce in each query, thereby completely eliminating the tag randomness from the authentication process. Moreover, we found the same weakness in the 32-bit random numbers generated by the reader chip, which suggests that a similar hardware implementation is used in the chip and reader. Here, too, we were able to repeatedly generate the same number. While in our experiments this meant controlling the timing of the reader chip, a skilled attacker will likely be able to exploit this vulnerability even in realistic scenarios where no such control over the reader is given. The attacker can predict forthcoming numbers from the numbers already seen and precisely chose the time to start interacting with the reader in order to receive a certain challenge. The lack of true randomness on both reader and tag enable an attacker to eliminate any form of randomness from the authentication protocol. Depending on the number of precomputed codebooks, this process might take several hours and the attack might not be feasible against all reader chips.

## 3.3 Pre-Computing Keys

Several weaknesses of the Mifare card design add up to what amounts to a full codebook pre-computation. First, the key space is small enough for all possible keys to be included. Second, the random numbers are controllable. In addition, the secret key and the tag ID are combined in such a way that for each session key there exists exactly one key for each ID that would result in that session key. The key and the ID are shifted into the register sequentially, but no non-linearity is mixed in during this process. As explained in Section 2.2, for every delta of ID bits, there exists a delta of key bits that corrects for the difference and results in the same session key. Therefore, given a key that for some ID results in a session key, there exists a key for any ID that would result in the same session key. This bijective mapping allows for a codebook that was pre-computed for only a single ID to be used to find keys for all other IDs as well.

A codebook for all keys would occupy 1500 Terabytes, but can be stored more economically in rainbow tables. Rainbow tables store just enough information from a key space for finding any key with high probability, but require much less space than a table for all keys [9, 15]. Each "rainbow" in these tables is the repeated application of slight variants of a cryptographic operation. In our case, we start with a random key and generate the output of the authentication protocol for this key, then use this output as the next key for the authentication, generate its output, use that as the next key, and so on. We then only store the first and last value of each rainbow, but compute enough rainbows so that almost all keys appear in one of them. To find a key from such a rainbow table, a new rainbow is computed starting at a recorded output from the authentication protocol. If any one of the generated values in this series is also found in the stored end values of the rainbows, then the key used in the authentication protocol can be found from the corresponding start values of that matching rainbow. The time needed to find a key grows as the size of the tables shrinks.

Determining any card's secret key will be significantly cheaper than trying out all possible keys even for rainbow tables that only occupy a few Terabytes and can be almost as cheap as a database lookup. The fact that an attacker can use a pre-computed codebook to reveal the keys from many cards dramatically changes the economics of an attack in favor of the attacker. This means that even attacks on low-value cards like bus tickets might be profitable.

## 3.4 Threat Summary

To summarize the threat to systems that rely on Mifare encryption for security, we illustrate a possible attack. An attacker would first scan the ID from a valid card. This number is unprotected and always sent in the clear. Next, the attacker would pretend to be that card to a legitimate reader, record the reader message of the challenge-response protocol with controlled random nonces, and abort the transaction. Given only two of these messages, the key of the card can be found in the pre-computed rainbow tables in a matter of minutes and then used to read the data from the card. This gives the attacker all the information needed to clone the card.

## 4 Discussion

The illustrated attack is yet another example of security-by-obscurity failing. Weaknesses in the exposed cipher reveal the pitfalls of proprietary cipher design without peer-review. A few changes in the design would have made some of the discussed attacks infeasible and could have increased the key size within the same hardware constraints to make brute-force attacks less likely. Much better security, however, can only be achieved through better, more thoroughly analyzed ciphers.

### 4.1 Potential Fixes

The system is vulnerable against codebook attacks because of its weak random numbers and the linear combination of key and ID. Both can be fixed without adding extra hardware or slowing down the operation.

Better, yet still not cryptographically sound, random numbers can be generated by exploiting the fact that memory cells are initially in an undetermined state [10]. The same behavior can be caused in flip-flops like those that make up the state register of the stream cipher simply by not resetting the flip-flops at initialization time. The cipher state would start in a random state and then evolve using the cipher's feedback loop until a random number is needed. At this point, the register contains a mostly unpredictable number of the size of the state register.

Because this design generates random numbers within the same registers that are used for the cipher states, it eliminates the need for a separate additional PRNG circuit. The saved area could then be spent on increasing the size of the cipher state. In the area of the 48-bit Crypto-1 and its 16-bit RNG, a 64-bit stream cipher that also produces significantly better pseudo-random number could hence be implemented. This increases the size and quality of the random numbers and at the same time increases the key size beyond the point where brute-force attacks can be done cheaply.

To further improve the resistance against codebook attacks, the non-linear feedback should be combined with either key or ID when shifted into the register to break the bijective mapping between different key-ID pairs. This measure does not increase implementation costs, since we only integrate the output of the filter function which is already computed.

To improve the resistance of the cipher against statistical attacks, the update function must be made non-linear, either by feeding some intermediate result of the filter function into the linear register or by using a non-linear feedback shift register instead.

None of the possible fixes will make the cipher appropriate for high security applications, but they improve the resistance against the most concerning attacks and can be done without any additional implementation cost.

### 4.2 Possible Defenses

Possible ways to protect against the described attacks include using standard, peer-reviewed, established cryptography such as the 3-DES block cipher that is already found on some of the more expensive cards including some of the Mifare line of products. A cheaper alternative that can be implemented in about twice the size of Crypto-1 is the Tiny Encryption Algorithm (TEA) [12, 18]. This established low-cost block cipher has publicly been scrutinized for several years and is so far only known to be vulnerable to some expensive attacks [11]. While TEA is far more secure than Crypto-1, it is also much slower. A Mifare authentication takes little more than one millisecond, while a minimum-size implementation of TEA would take about ten times as long. This would still be fast enough for most applications where Mifare cards are currently used.

Other known ways to protect against card cloning include fraud detection algorithms that are widely used in monitoring credit card transactions. These algorithms detect unusual behavior and can prevent fraudulent transactions, but require storing and analyzing transaction data, which runs contrary to the desire for privacy in RFID applications. Fraud protection systems also require all readers to be constantly connected to a central server, which is not the case in some of the current and planned deployments of RFID tags where offline readers are used.

Tamper-proofing can be used to protect secret keys from attackers, but provides little help against hardware reverse-engineering because the structure of the circuits will always be preserved. The implementation, however, could be obfuscated to increase the complexity of the circuit detection. While we believe that obfuscations will not make our approach infeasible, we do not yet know to what degree obfuscations could increase the effort and cost required to reverse-engineer a circuit.

All low-cost cryptographic RFID tags are currently ill-suited for high security applications because they lack tamper-proofing and are vulnerable to relay attacks. In these attacks, the communication between a legitimate reader and a valid card is relayed through a tunnel thereby giving the reader the false impression that the card is in its vicinity. No level of encryption can protect against relay attacks and new approaches such as distance bounding protocols are needed [8].

## 5 Conclusions

Reverse-engineering functionality from silicon implementations can be done cheaply, and can be automated to the point where even large chips are potential targets. This work demonstrates that the cost of finding the algorithm used in a hardware implementation is much lower than previously thought. Using template matching, algorithms can be recovered whose secrecy has so far provided a base for security claims. The security of embedded cryptography, therefore, must not rely on obscurity. Any algorithm given to users in form of hardware can be disclosed even when no software implementation exists and black-box analysis is infeasible. Once the details of a cryptographic cipher become public, its security must rely entirely on good cryptographic design and sufficiently long secret keys.

The cryptographic strength of any security system depends on its weakest link. Besides the cryptographic structure of the cipher, weaknesses can arise from protocol flaws, weak random numbers, or side channels. When random numbers are weak and the user identification is not properly mixed into the secret state, codebooks can be pre-computed that lead to attacks that are much more efficient than brute force. In the case of the Mifare Classic cards, the average attack cost shrinks from several hours to minutes. Their cryptographic protection is hence insufficient even for low-valued transactions.

The question remains open as to whether security can be achieved within the size of the Mifare Crypto-1 cipher. The area of less than 500 gates may be too small to even hold a sufficiently large state, regardless of the circuits needed for the complex operations required for strong ciphers.

**Acknowledgments.** This work was partially funded by the National Science Foundation through the CyberTrust program, award CNS 0627527. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect those of the National Science Foundation.

## References

[1] Ross Anderson. *A5*. Post to *sci.crypt*, 17 June 1994.

[2] L. R. Avery, J. S. Crabbe, S. Al Sofi, H. Ahmed, J. R. A. Cleaver, D. J. Weaver. Reverse Engineering Complex Application-Specific Integrated Circuits (ASICs). In *Diminishing Manufacturing Sources and Material Shortages Conference*, 2002.

[3] Andrey Bogdanov. Attacks on the KeeLoq Block Cipher and Authentication Systems. In *RFIDSec*, 2007.

[4] Stephen C. Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D. Rubin, and Michael Szydlo. Security Analysis of a Cryptographically-Enabled RFID Device. In *USENIX Security Symposium*, 2005.

[5] Harvey G. Cragon. *From Fish to Colossus: How the German Lorenz Cipher was Broken at Bletchley Park*. Cragon Books, 2003.

[6] Electronic Frontier Foundation. Cracking DES. In *Secrets of Encryption Research, Wiretap Politics & Chip Design*, O'Reilly & Associates Inc., 1998.

[7] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong Authentication for RFID Systems using the AES Algorithm. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[8] Gerhard P. Hancke and Markus G. Kuhn. An RFID Distance Bounding Protocol. In *SecureComm*, 2005.

[9] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. In *IEEE Transactions on Information Theory*, 1980.

[10] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial SRAM state as a Fingerprint and Source of True Random Numbers for RFID Tags. In *RFIDSec*, 2007.

[11] Seokhie Hong, Deukjo Hong, Youngdai Ko, Donghoon Chang, Wonil Lee, and Sangjin Lee. Differential Cryptanalysis of TEA and XTEA. In *International Conference on Information Security and Cryptology*, 2003.

[12] Pasin Israsena. Securing Ubiquitous and Low-cost RFID Using Tiny Encryption Algorithm. In *International Symposium on Wireless Pervasive Computing*, 2006.

[13] Auguste Kerckhoffs. La Cryptographie Militaire. In *Journal des Sciences Militaires*,

1883.

[14] J. P. Lewis. Fast Normalized Cross-Correlation. In *Vison Interface*, 1995.

[15] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Crypto*, 2003.

[16] NXP Semiconductors. *Philips Semiconductors leads contactless smart card market*, 2006.

[17] Andrew Tanenbaum. *News Summary of Broken Dutch Public Transit Card*. `www.cs.vu.nl/~ast/ov-chip-card`

[18] David J. Wheeler and Roger M. Needham. TEA, a Tiny Encryption Algorithm. In *Fast Software Encryption*, 1994.

# Practical Symmetric Key Cryptography on Modern Graphics Hardware

Owen Harrison
*Computer Architecture Group*
*Trinity College Dublin*
*Dublin 2, Ireland*
*harrisoo@cs.tcd.ie*

John Waldron
*Computer Architecture Group*
*Trinity College Dublin*
*Dublin 2, Ireland*
*john.waldron@cs.tcd.ie*

## Abstract

Graphics processors are continuing their trend of vastly outperforming CPUs while becoming more general purpose. The latest generation of graphics processors have introduced the ability handle integers natively. This has increased the GPU's applicability to many fields, especially cryptography. This paper presents an application oriented approach to block cipher processing on GPUs. A new block based conventional implementation of AES on an Nvidia G80 is shown with 4-10x speed improvements over CPU implementations and 2-4x speed increase over the previous fastest AES GPU implementation. We outline a general purpose data structure for representing cryptographic client requests which is suitable for execution on a GPU. We explore the issues related to the mapping of this general structure to the GPU. Finally we present the first analysis of the main encryption modes of operation on a GPU, showing the performance and behavioural implications of executing these modes under the outlined general purpose data model. Our AES implementation is used as the underlying block cipher to show the overhead of moving from an optimised hardcoded approach to a generalised one.

## 1 Introduction

With the introduction of the latest generation of graphics processors, which include integer and float capable processing units, there has been intensifying interest both in industry and academia to use these devices for non graphical purposes. This interest comes from the high potential processing power and memory bandwidth that these processors offer. The gap in processing power between conventional CPUs and GPUs (Graphics Processing Units) is due to the CPU being optimised for the execution of serial processes with the inclusion of large caches and complex instruction sets and decode stages. The GPU uses more of its transistor budget on execution units rather than caching and control. For applications that suit the GPU structure, those with high arithmetic intensity and parallelisability, the performance gains over conventional CPUs can be large. Another factor in the growth of interest in general purpose processing on GPUs is the provision of more uniform programming APIs by both major graphics processor vendors, Nvidia with CUDA (Compute Unified Device Architecture) [1] and AMD with CTM (Close To Metal) [2].

The main obstacle with achieving good performance on a GPU processor is to ensure that all processing units are busy executing instructions. This becomes a challenge in consideration of Nvidia's latest processor, which contains 128 execution units, given the restrictions of its SPMD (Single Program Multiple Data) programming model and the requirement to hide memory latency with a large number of threads. With respect to private key cryptography and its practical use, a challenge exists in achieving high efficiency particularly when processing modes of operation that are serial in nature. Another practical consideration is the current development overhead associated with using a GPU for cryptographic acceleration. Client applications would benefit from the ability to map their general cryptographic requirements onto GPUs in an easy manner.

In this paper we present a data model for encapsulating cryptographic functions which is suitable for use with the GPU. The application of this data model and the details of its interaction with the underlying GPU implementations are outlined. In particular we investigate how the input data can be mapped to the threading model of the GPU for modes of operation that are serial and parallel in nature. We show the performance of these modes and use our optimised AES implementation to determine the overhead associated with using a flexible data model and its mapping to the GPU. Also included in the paper is a study of the issues related to the mixing of modes of operation within a single GPU call.

**Motivation:** The motivation for this research is based on the GPU acting as a general guide for the long-term direction of general purpose processing. X86 architectures are bottlenecking with limited increase in clock frequency reported in recent years. This is being tackled by the addition of cores to a single die to provide growth in the total available clock cycles. The GPU is the logical extreme of this approach where the emphasis has always been on more but simpler processing elements. The upcoming AMD's Accelerated Processing Unit (Swift) [3] architecture is a reasonable compromise where a CPU and GPU are combined onto a single chip. Also Intel are developing computing solutions under the TeraScale banner which include a prototype of an 80 core processor. Using GPUs as a research platform exposes the issues that general purpose processing will encounter in future highly parallel architectures. Another motivation is the use of GPUs as a cryptographic co-processor. The types of applications that would most likely benefit are those within a server environment requiring bulk cryptographic processing, such as secure backup/restore or high bandwidth media streaming. We also wish to show the implications of the inclusion of the GPU as a generic private key cryptographic service for general application use.

**Organisation:** In Section 2 a brief description of the essentials in GPU hardware used is outlined, along with the CUDA programming model. Section 3 shows the related work in cryptography on non general purpose processors with a focus on GPUs. We present an implementation of AES on the Nvidia's G80 architecture and show its performance improvements over comparable CPU and GPU implementations in Section 4. In Section 5 we introduce the generic data model suited to GPUs, which is used to encapsulate application cryptographic requirements. Section 6 describes in detail the steps of mapping from the generic data structure to underlying GPU implementations. All three previous sections are combined by the implementation of modes of operation using the outlined data model and the optimised AES implementation in Section 7. This shows the overheads associated going from a hardcoded to a more general purpose implementation.

## 2 GPU Background

In this section we present a brief account of the GPU architecture used in the implementations presented within this paper, the Nvidia G80. We also give an outline of the new CUDA [1] programming model which has been introduced by Nvidia to provide a non graphics API method of programming the G80 generation of processors. Previous to this programming interface either OpenGL [4] or DirectX [5] had to be used at a considerable learning expense to the programmer. AMD have also introduced their own software stack to tackle the issue of providing a more user friendly programming interface to their processors - CTM [2], however we do not cover this here. The G80 processors are DX10 [6] standard compliant which implies it belongs to the first generation of GPUs which support integer data units and bitwise operations. A key advancement relating to the field of cryptography.

**Physical View:** The G80 can consist of up to 16 multiprocessors within a single chip. Each of these multiprocessors consist of 8 ALU (Arithmetic and Logic Unit) units which are controlled by a single instruction unit in a SIMD (Single Instruction Multiple Data) fashion. The instruction unit only issues a single instruction to the ALUs every four clock cycles. This creates an effective 32 SIMD width for each multiprocessor, ie. a single instruction for 32 units of data. Each multiprocessor has limited fast on-chip memory consisting of 32 bit register memory, shared memory, constant cache and texture cache. All other forms of memory, linear, texture arrays are stored in global memory, ie. off-chip. GPUs can be used in arrangements of multiple chips on a single graphics card and also multiple boards on a single mother board. For all implementations and comparisons with CPUs we have restricted the arrangements used to single GPU and single CPU core.

**Execution Model:** The CUDA programming model provides a way to programme the above chip in a relatively straight forward manner. The programmer can define threads which run on the G80 in parallel using standard instructions we are familiar with within the field of general purpose programming. The programmer declares the number of threads which must be run on a single multiprocessor by specifying a block size. The programmer also defines multiple blocks of threads by declaring a grid size. A grid of threads makes up a single kernel of work which can be sent to the GPU and when finished, in its entirety, is sent back to the host and made available to the CUDA application.

Two more points of note which are relevant to this paper. First, all threads within a single block will run only on a single multiprocessor. This allows threads within a single block to have the ability to share data with other threads within the block via shared memory. Inter block communication is not possible as there is no synchronisation method provided for this. Second, due to the 32 SIMD wide execution arrangement described above, Nvidia have introduced the notion of a warp. A warp represents a grouping of threads into a unit of 32. These threads run on the same multiprocessor for the entire 4 cycles required to execute a single instruction. Threads are assigned to a warp in a simple serially increasing order starting a 0 for the first thread within a block. Performance issues can arise when a group of 32 threads diverge in their code path, this causes the entire 4 cycles to be run for every unique instruction required by the 32 threads.

## 3 Related Work

A variety of non general purpose processors has been used in the implementation of private key ciphers over the years. Specifically within the field of graphics processors, the first implementation of any cipher was by Cook et al. [7]. They implemented AES on an Nvidia Geforce3 Ti200, which had little programmable functionality. Their implementation was restricted to using the OpengGl library and only a fixed function graphics pipeline. They describe the use of configurable color maps to support byte transforms and the use of the final output stage of the pipeline (Raster Operations Unit (ROP)) to perform XORs. Unfortunately due to the restrictive nature of the hardware used and having to perform all XORs in the final output stage of the pipeline, multiple passes of the pipeline were required for each block. The authors presented a successful full implementation running within the range of 184 Kbps - 1.53 Mbps.

Harrison et al. [8] presented the first CPU competitive implementation of a block cipher on a GPU. They used the latest DX9 compliant generation of graphics processor to implement AES, namely an Nvidia 7900GT. These processors support a more flexible programming model compared to previous models, whereby certain stages of the graphics pipeline can execute C like programmer defined threads. However, the 7900GT only supports floating point operations. 3 different approaches were investigated to overcome the lack of integer bitwise operations on the programmable portion of the pipeline. The XOR operation was simulated using lookup tables for 4 bit and 8 bit XORs, and also the hardware supported XOR function within the final stage (ROP) of the pipeline. Their results showed that a multipass implementation using the built in XOR function combined with a technique called ping-ponging of texture memory to avoid excess data transfers across the PCIe bus could be used to achieve a rate of 870 Mbps.

More recently the latest generation of hardware, which supports integer data types and bitwise operations, has been used by Yang et al. [9] to produce much improved performance results. This paper focuses on a bitslicing implementation of DES and AES which takes advantage of the AMD HD 2900 XT GPU's large register size. The GPU is used as a 4 way 32 bit processor which operates on four columns of 32 bitsliced AES state arrays in parallel. They show rates of 18.5 Gbps processing throughput for this bitsliced AES implementation. A bitsliced implementation isn't suitable for general purpose use as it requires heavy preprocessing of the input blocks. The authors [9] argue that their bitslicing approach can be put to use as a component in template-based key searching utility or for finding missing key bytes in side channel attacks whereby the input state is static relative to the key. A conventional block based implementation of AES is also presented in this paper, running at rates of 3.5 Gbps. Whether this includes transfers of input/output blocks across the PCIe bus is not indicated.

Other non general purpose processors used for private key cryptography include various ASIC designs such as [11] [12] [13] and custom FPGA efforts [14] [15]. GPUs have also been applied in the field of public key cryptography. A paper by A. Moss et al. [10] tackles the problem of executing modular exponentiation on an Nvidia 7800 GTX. They present promising results showing a speed up of 3 times when compared to a similar exponentiation implementation on a standard X86 processor. Also related to graphics processors, Costigan and Scott [16] presented an implementation of RSA using the Playstation's 3 IBM Cell processor. They were able to increase the performance of RSA using the Cell's 8 SPUs over its single PowerPC core.

## 4 Block Based AES Implementation

In this section we present an optimised implementation of the AES cipher in CTR mode on an Nvida 8800 GTX (G80) using the CUDA programming model. The aim of this section is to provide the performance figures and implementation approach which will be used in conjunction with the data model described in Section 5. As such the chosen implementation in this section is an ideal, non general purpose, implementation which can be used as a source of comparison with generalised approaches.

As previously mentioned the G80 architecture supports integer bitwise operations and 32 bit integer data types. These new features, which are shared by all DX10 [6] compatible GPUs, simplify the implementation of AES and other block ciphers. This allows for a more conventional AES approach compared to implementations on previous generations of graphics processors. We based our implementations around both the single 1 KB and 4 x 1 KB precalculated lookup tables which were presented in the AES specification paper [17], see Equations 1 and 2 respectively.

$$\begin{aligned} e_j &= k_j \oplus T_0[a_{(0,j)}] \oplus Rot(T_0[a_{(1,j-c1)}] \oplus \\ &\quad Rot(T_0[a_{(2,j-c2)}] \oplus Rot(T_0[a_{(3,j-c3)}]))) \; . \end{aligned} \tag{1}$$

$$\begin{aligned} e_j &= T_0[a_{(0,j)}] \oplus T_1[a_{(1,j-c1)}] \oplus T_2[a_{(2,j-c2)}] \\ &\quad \oplus T_3[a_{(3,j-c3)}] \oplus k_j \; . \end{aligned} \tag{2}$$

As XORs are supported in the programmable section of the graphics pipeline, there is no need to use the ROP XOR support, which required multiple passes of the pipeline - one for each XOR operation. Each thread that is created, calculates its own input and output address for a single data block and runs largely in isolation of other threads in a single pass to generate its results. The simple thread to I/O data mapping scheme used for all implementations reported in this section is as follows. Each thread's index relative to the global thread environment for a kernel execution is used as the thread's offset into the input and output data buffers:

```
int index = threadIdx.x + (blockIdx.x * blockDim.x);
uint4 state = pt[index];
ct[index] = state;
```

where blockDim is the number of CUDA blocks within the CUDA grid, blockId is the current CUDA block the thread exists within and threadId is the current thread index within the CUDA block. As CTR is a parallel mode of operation, each thread works on a single AES block independently of other threads. To achieve high performance on a GPU or any highly multi-threaded processor, an important programming goal is to increase occupancy. The level of occupancy on a parallel processor indicates the number of threads available to the thread scheduler at any one time. High occupancy ensures good resource utilisation and also helps hide memory access latency. It is for occupancy reasons that we create a single thread for each input block of data.

A nonce is passed to all threads through constant memory and the counter is calculated in the same manner as the data offsets above. Rekeying was simplified by using a single key for all data to be encrypted, with the key schedule generated on the CPU. The reason for implementing the rekeying process on the CPU rather than the GPU is that it is serial in nature, thus the generation of a key schedule must be done within a single thread. It would be an unacceptable overhead per thread (ie. per data block) when processing a parallel mode of operation, for each thread to generate its own schedule.

**Host and Device Memory:** We investigated using both textures and linear global memory to store the input and output data on the device. Through experimentation we found global memory to be slightly faster than texture memory for input data reads and writes, thus all our implementation results are based on using linear global memory reads and writes for plaintext and ciphertext data. Regarding host memory (CPU side), an important factor in performance of transferring data to and from the GPU is whether one uses page locked memory or not. Page locked memory is substantially faster than non page locked memory as it can be used directly by the GPU via DMA (Direct Memory Access). The disadvantage is that systems have limited amount of page locked memory as it cannot be swapped, though this is seen normally as an essential feature for secure applications to avoid paging sensitive information to disk.

| | Coherent Reads | Random Reads |
|---|---|---|
| Shared Memory | 0.204319s | 0.433328s |
| Constant Memory | 0.176087s | 0.960423s |
| Texture Memory | 0.702573s | 1.237914s |

Table 1: On-chip Memory Reads: Average execution times of 5 billion 32-bit reads.

**On-chip Memory:** As the main performance bottleneck for a table lookup based AES approach is the speed of access to the lookup tables, we implemented both lookup table versions using all available types of on-chip memory for the G80. The types used are texture cache, constant cache and shared memory. Shared memory is shared between threads in a CUDA block and is limited to 16 KB of memory per multiprocessor. It should be noted that shared memory is divided into 16 banks, where memory locations are striped across the banks in units of 32 bits. 16 parallel reads are supported if no bank conflicts occur and for those that do occur, they must be resolved serially. The constant memory cache working set is 8 KB per multiprocessor and single ported, thus it only supports a single memory request at one time. Texture memory cache is used for all texture reads and is 8 KB in size per multiprocessor. To investigate the the read performance characteristics of these types of memory we devised read tests to access the three types of memory in two different ways. We split the tests into random and coherent read memory access patterns, each test accessing 5 billion integers per kernel execution. Coherent access patterns were included as there are opportunities to exploit coherent reads within shared memory, ie. reads with no bank conflicts for a half warp of 16 threads.

In Table 1 we can see the average execution times measured in seconds to perform the 5 billion reads. Constant memory performs best with regard to coherent reads, though as constant memory is single ported and the lookup tables will be accessed randomly within a warp it is of little use. Shared memory out performs in the scenario of random access reads by a large margin due to its high number of ports. It should be noted that both texture and constant memory can be loaded with data before the kernel is called, however a disadvantage to shared memory is that it must be setup once per CUDA block. Shared memory is designed for use as an inter thread communication memory within the one multiprocessor and not designed for preloading of data. This is most likely an API limitation and would help reduce the setup overhead if a mechanism existed to setup shared memory once per

|  | Shared Memory | Constant Memory | Texture Memory |
|---|---|---|---|
| Single Table | 5,945 Mbps | 4,123 Mbps | 4,200 Mbps |
| Quad Table | 6,914 Mbps | 4,085 Mbps | 4,197 Mbps |

Table 2: AES CTR maximum throughput rate for different types of on-chip memory.

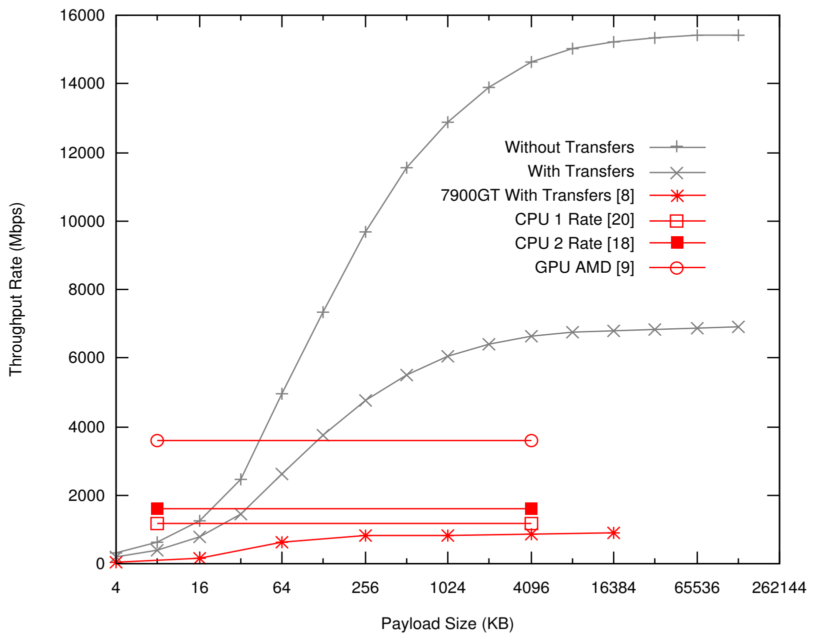


Figure 1: Optimised AES CTR implementation with and without data transfers.

multiprocessor before kernel execution. In an attempt to gain from the performance benefits of coherent memory reads when using shared memory we copied a single lookup table 16 times across all 16 banks to avoid memory conflicts with careful memory addressing. However it turns out that CUDA does not allow access to all 16K, even though it advertises it as such. In fact the developer only has access to slightly less than 16 KB, as the first 28 bytes are reserved for system use and if over written by force causes system instability. Various optimisations were attempted to avoid bank conflicts, the fastest approach used 16 x 1 KB tables save the last entry. A simple check if the last lookup entry is being sought and its direct value is used instead.

**AES:** In Table 2 we can see the maximum performance of the different AES implementations using the different types of on-chip memory. It can be seen that the 4 x 1 KB table approach, Quad Table, using shared memory performs the fastest. This approach requires the four 1 KB tables to be setup within shared memory for each CUDA block of threads running. This setup can be alleviated by allocating the task of a single load from global memory into shared memory to each thread within the block. For this reason our implementation uses 256 threads per block, giving the least amount of overhead to perform the setup operation. The coherent shared memory 1 KB table lookup under performs due to the extra rotates which must be executed, the extra conditional check for sourcing the last table entry as described above and the additional per CUDA block memory setup costs. Previous generations of GPUs could hide the cost of state rotates via the use of swizzling (the ability to arbitrarily access vector register components) however the G80 no longer supports this feature.

Figure 1 shows the performance of AES CTR based on the above 4 x 1 KB table lookup approach. The figure exposes the requirement of many threads to hide memory read latency within the GPU. We display the throughput rate of the cipher with and without plaintext and ciphertext transfers across the PCIe bus per kernel execution. A maximum rate of 15,423 Mbps was recorded without transfers and a maximum of 6,914 Mbps was recorded with transfers included. We have included the rates without data transfer as we believe these to be relevant going forward where the possibility exists for either: sharing the main memory address space with the CPU, either in the form of a combined processor or a direct motherboard processor slot; or overlapping kernel execution and data transfer support on the GPU.

In the same figure we have compared our results with the latest reported AES implementations on CPUs and GPUs. Matsui [20], reports speeds of 1,583 Mbps for conventional block based implementation of AES on a 2.2 GHz AMD 64 processor. [18] reports an ECB AES implementation of 1,151Mbps on an AMD 64 2.4 GHz 64 processor. The authors of [9], cite a speed of 3,584 Mbps on an AMD HD 2900 XT GPU (AMD's DX10 compliant GPU) for their block based AES implementation, though we do not have access to the throughput rates as data sizes increase. We have included the rates achieved in [8] for AES on a GeForce 7900GT, which does provide this rate progression. With transfers included, we see a 4x speed up over the fastest CPU reported implementation and a 10x speed up without transfers. Scaling up the reported CPU AES rates to the latest available AMD core clock speed, our GPU implementation still substantially outperforms. When compared to the block based AES implementation on a GPU by [9] we can see 2x and 4x speeds ups with and without data transfers respectively.

## 5 Payload Data Model

In this section we introduce the generic data model which we use to allow the exploration of the problems involved in mapping a generic private key cryptographic service to specific GPU implementations. The aim of this section is to outline the data model used, its design criteria and the usage implications in the context of GPUs.

### 5.1 The Data Model

We use the term payload to indicate a single grouping of data which contains both data for processing and its instructions. The client application which requires cryptographic work is responsible for the creation of a payload and hand off to a runtime library which can direct the payload to the appropriate implementation. The data model described is similar to the fundamental principals of the OpenBSD Cryptographic Framework [19] and as such the implementations presented could potentially be integrated into such a runtime environment.

One of the main criterion for a data model in this context is to allow the buffering of as many messages as possible that require processing into a single stream, permitting the GPU to reach its full performance potential. Exposing a payload structure to the user rather than a per message API allows the grouping of multiple messages. Also, the pressure for increase in data size can be met by providing the ability for different cryptographic functions to be combined into a single payload. Another design criteria is the use of offsets into the data for processing rather than pointers as the data will be transferred outside of the client applications memory address space rendering pointers invalid. The data model must also allow the client to describe the underlying data and keys, with key reuse, in a straight forward manner.

In the following pseudocode we can see the key data structures used. The main "payload" structure contains separate pointers for data and keys as these are normally maintained separately. A single payload descriptor structure is also referenced which is used to describe the mapping of messages to the data and key streams. The payload descriptor uses an ID to uniquely identify payloads in an asynchronous runtime environment. A high level mode for which cryptographic service is required, can be described within the payload descriptor or within the individual messages. The need for a higher level mode is due to the requirement of frameworks which abstract from multiple hardware devices having to select suitable hardware configuration to implement the entire payload. A lower level property can also be used to describe the cryptographic mode on a per message basis as can be seen in the "msgDscr" structure. The message descriptor also provides pointers for arrays of messages, IVs (Initialisation Vector), ADs (Associated Data), tags, etc. Each of these elements use the generic element descriptor which allows the description of any data unit within the data and key stream using address independent offsets. The element descriptor separates the concept of element size and count as the size of elements can sometimes indicate a functional difference in the used cipher. The return payload is the similar to the payload structure, though without the keys.

```
struct payload {
    unsigned char *data;
    unsigned char *keys;
    struct payloadDscr *dscr; };

struct payloadDscr {
    unsigned int id;
    struct keyValue *payloadMode;
    unsigned int msgcount;
    unsigned int size;
    struct msgDscr *msgs;
    struct elementDscr *keys; };

struct msgDscr {
    struct element_dscr *msg;
    struct element_dscr *iv;
    struct element_dscr *ad;
    struct element_dscr *tag;
    struct key_value **msgMode; };

struct element_dscr {
    unsigned int count;
    unsigned int offset;
    unsigned int size; };
```

### 5.2 General Use Implications

A consideration regarding the use of per message properties to indicate separate functions is that it adds extra

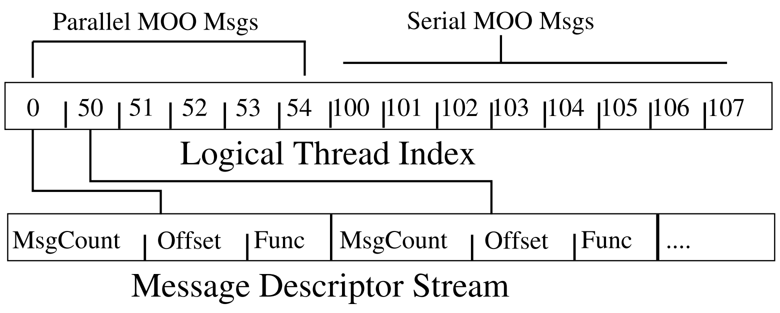


Figure 2: Serialised Streams used by each thread for Data and Key indexing.

register pressures on SPMD architectures such as GPUs. These processors can only execute a single kernel code across all threads, any variation in function must be implemented using conditional branches. This technique is called using fat kernels, where a conditional branch indicates a large variation in underlying code executed at run time. On SPMD processors it is better for performance if all messages within a payload use the same function, which is determined before kernel execution time.

Another concern when employing a data model for use with an attached processor, such as a GPU, is memory allocation for I/O buffers. For the G80 it is important to use pinned (page locked memory), this requires a request to be made to the CUDA library. The CUDA library then returns a pointer to the memory requested which can be used within the calling process. Both the input and output buffers should use pinned memory and also reuse the same buffers when possible for maximum performance. Thus there is a need for the client to be able to request both input and output buffers, to allow the tracking of its allocated buffers as the implementation cannot make a buffer reuse decision independently. This requires the encapsulating runtime, for example such as a framework like the OpenBSD Cryptoraphic Framework, to support mapping of memory allocation requests through to the library representing the hardware which will service the payload.

## 6 Applied Data Model

In this section we cover implementation concerns when bridging between the previously described general purpose data model and specific GPU cipher implementations. In particular we focus on our implementation of a bridging layer, which maps the data model to our specific cipher modes of operation presented in Section 7. The overhead of providing a general purpose interface point to a GPU implementation is the addition of abstraction layers which need to be resolved within each kernel thread. Throughput is lost when message functions, sizes, element types, etc can vary within a payload. Each thread must perform extra memory accesses, calculations and conditional branches to dereference these dynamic settings. These per thread calculations can be offset by an implementation using the CPU as a preprocessing stage which optimises a payload for thread parsing before the payload is dispatched. Naturally there is a balance to CPU preprocessing, as one of the reasons for using a GPU is to act as a co-processor which in effect speeds up the overall throughput of a system.

### 6.1 Descriptor Serialisation

Each element in the message descriptor requires serialising on the CPU into a form which can be used independently and quickly within each thread on the GPU. An implementation determines the message descriptor element size during serialisation, thus given a message ID, a thread can directly lookup the corresponding message instructions. Each serialised element contains the message data stream offset, size, function, and whatever other information is required specific to the implementation. The key descriptor, which contains the access information for the key schedules, requires the generation of a separate key schedule stream before it can be serialized. Both serialised descriptor streams and the key schedule streams are transferred to the GPU and stored within texture memory address space, which gives the best size flexibility of the cacheable memory types.

**Logical Thread Index:** During message serialisation a logical thread index stream is produced to facilitate the efficient location of a message ID given a thread's ID. This stream contains a single thread for each serial mode of operation (MOO) message and as many threads as there are blocks for each parallel MOO message. The entries within the logical thread index consist of only the logical thread IDs which start a message. Figure 2 shows an example of a logical thread index and how it relates to the message descriptor stream. We call the stream a **logical** thread index because the physical thread IDs (those assigned by the GPU), which partially determine the thread's physical location within the processor, do

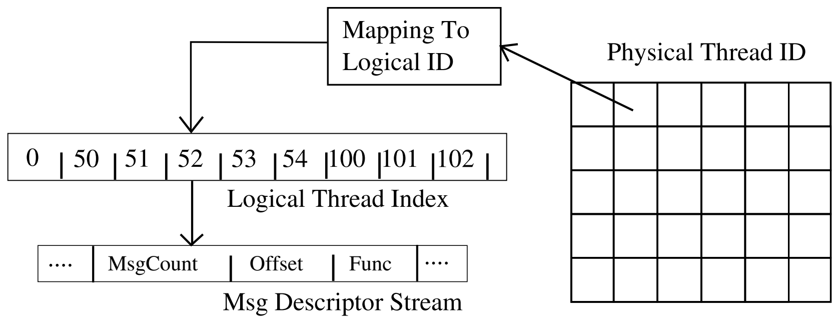


Figure 3: Process of mapping physical threads to message IDs.

not necessarily map directly onto the entries within the thread index. To support balancing of work across the multiprocessors of the GPU we require the ability to assign work to different threads depending on their physical ID. Balancing work across the GPU is important for serial MOO messages, where the number of messages may be low and the size of messages may be high.

**Rekeying:** As outlined previously, the GPU is a highly parallel device and the key schedule generation is inherently serial, thus in general it makes most sense to implement keying on the CPU prior to payload dispatch. Our implementation uses the CPU with a hashtable cache for storing key schedules to ensure key reuse across messages. This is not just to aid efficiency at the key schedule generation stage on the CPU but also to generate the smallest key schedule stream possible. This is important for on-chip GPU caching of the key schedules. When the client application is generating the key stream for payload inclusion, it is important for the same keys to use the same position within the stream. This allows for fast optimisation of key schedule caching based on key offsets rather than key comparison.

## 6.2 Thread to Message Mapping

The full process for mapping a thread to a message ID and its underlying data is the following, this is also shown in simplified form in Figure 3.

**1.** Generation of the logical thread index for all messages as outlined previously. This work is carried on on the CPU.

**2.** Mapping of the physical (GPU assigned) thread ID to a logical thread ID within each kernel thread. This was implemented using two different algorithms, one with a focus on performance and the other a focus on client application control for load balancing. The first approach maps physical to logical threads in a 1 to 1 manner in multiples of 16 x 256 threads, until the last and potentially partially full 16 x 256 group of threads. The last group of threads is allocated evenly across the multiprocessors assigning physical IDs in natural order. The reason for using 16 x 256 threads, is that the implementations used assign a fixed 256 threads per CUDA block in multiples of 16 blocks (ie. the number of multiprocessors on the 8800GTX processor). This is done to ensure the simplest form of shared memory configuration for lookup tables, see Section 4. This approach is fast to execute as a single check can eliminate the case of full thread groups where physical and logical IDs are the same. The second approach maps each physical thread into groups of 32 striped across each CUDA block. This mapping is executed for every thread and thus is slightly slower than the first approach. It however gives a more consistent mechanism for mapping physical threads to messages and thus is more controllable by the client application. In the first approach its difficult or impossible to insert serial MOO messages so that they are evenly spread across the available multiprocessors. We use the second approach in our reporting of results as it is only 0.25% slower than the first and thus the advantage out weighs the performance hit. See Section 7.3 for the effects of loadbalancing work across the GPU.

**3.** Search of logical thread index with logical thread ID to determine message ID for kernel. This step also calculates logical thread ID offset from beginning of message. Due to storing a digested form of the logical thread IDs, in which each entry in the logical thread index is the thread ID start of a message, the search is implemented as a binary search. A direct lookup table could be used for better performance however this would require a lookup table equal to the number of logical threads. For parallel MOO messages this would be too high an overhead in terms of data transfer and cacheability of the index. The digest version only stores a thread index entry per message within the payload and also provides an easy way to calculate the thread offset from the start thread (ie. first block) for parallel MOOs messages.

**4.** Use of message ID to offset into the message descriptor stream, which is used to retrieve the input data offset and other message settings.

## 6.3 Padding

The client application can set padding or not for each message within the message descriptor. As the ability to generate a link list of addresses for use during DMA transfer is not supported in CUDA, it results in too high an overhead for the CPU based serialization process to support pre-padding message directly into the data stream for sending to the attached device. The reason for this is that the CPU would have to generate a new single stream from contiguous memory based on the new insertions and the original data stream. An alternative more efficient approach is to embedded the padding instructions into the message descriptor stream which indicates the types of padding required. This requires that each thread checks if extra padding is required and to generate the necessary extra data itself. In relation to CUDA this extra check causes thread divergence for the single thread that must execute the padding. However the overhead is generally very low as the divergence only lasts for a single cipher block across 32 threads. If a full new block is required, as potentially in PKCS#5 for example, then an extra block is required in the output. This is an issue for GPUs as typically a live thread cannot allocate its own memory. The CPU must allocate for this extra space during serialisation before the payload is sent to the GPU.

## 6.4 Payload Combining

The bridging layer implementation can easily implement payload combining in the scenario where payloads are queued via the encapsulating framework. The multiple data and key schedule streams within host memory space can be copied into consolidated input buffers on the attached device. During serialisation stage, the serialised message and key descriptors are appended and offsets are recalculated taking into account the combined input streams on the attached device. Similarly processed payloads can be read from a consolidated output buffer on the attached device and read into separate host buffers. Generally ciphers do not change the size of the plaintext and ciphertext, padding aside, allowing efficient reuse (directly or copies) of the input payload descriptors.

## 7 Modes Of Operation

In this section we present the implementation and results of symmetric key modes of operation built using the previously described data model, bridging layer and AES implementation. Modes of operation determine how the underlying block cipher is used to implement a cryptographic system which supports messages greater than one block in length. We have analysed the most common encryption modes, specifically CTR, CBC, CFB and OFB. Using these modes on a highly multithreaded device, the major overriding characteristic which determines throughput is whether the mode can be implemented in parallel or must be done serially. CFB's latency reduction is not relevant within the context of a payload where the entire message is sent and read back as a single unit. OFB and CTR allow the pregeneration of a key stream with subsequent XORing with the plaintext/ciphertext for its operation. This can provide good latency reduction whereby the execution of a payload can be split into two separate stages, one for key stream generation and one for XORing. However, regarding an application that will gain from the use of a bulk cryptographic co-processor, the most important characteristic is throughput. We focus on the throughput of the two main categories of MOOs: serial MOO (CBC and CFB encryption and OFB), and parallel MOO (CBC and CFB decryption and CTR). All implementations are based on the optimised AES implementation presented in Section 4 using CUDA. Discounting block cipher performance variation, these results should provide a guide to the general behaviour of the investigated MOOs using other block ciphers on a GPU.

## 7.1 Parallel MOOs

It is easier to achieve full occupancy on a highly parallel processor such as a GPU when processing parallel MOO messages compared to serial MOO messages. Each message can be split into blocks and assigned its own thread, thus the number of threads equals the total number of blocks within the payload. Figure 4 shows the throughput rates of different message sizes used within payloads containing parallel MOO messages. The results shown are based on the CTR MOO. CBC and CFB decryption were also implemented, though the throughput rates did not vary. The number of messages indicates the number used within a single payload. As we can see, the greater the payload size the higher the performance. This is expected due to increased resource occupancy and memory latency being more effectively hidden. We can also see that at a certain throughput rate the per message overhead of using a generic data model becomes the dominant overhead. As a result, increasing the payload message count past a certain point results in a drop in performance.

All results are based on multiple executions of a single payload with the reuse of memory buffers both for host and on device storage. This simulates the scenario of an application managing its own host memory allocations as described in Section 5. Our implementations also reuse the same key, simulating all messages being within a single cryptographic session. We also include

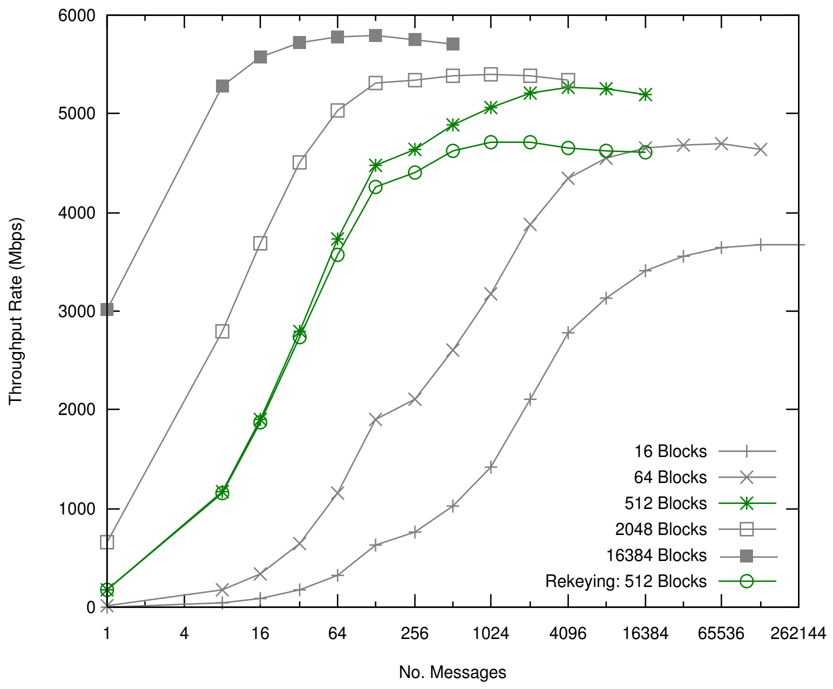


Figure 4: Throughput rates for parallel MOO messages across varying block sizes.

in Figure 4 throughput rates for maximum rate rekeying, whereby each message contains its own unique key. We have highlighted the comparison of rates with and without rekeying for payloads with a message size of 512 blocks. As expected, an increasing message count results in an increasing overhead on total throughput.

The maximum throughput achieved for a parallel MOO under the generic data model was 5,810 Mbps. An important observation from these figures is that we can see there is an overhead associated with using the described generic data model for abstracting the underlying implementation details. When using large messages (16384 blocks) this overhead is 16%, with medium sized messages (512 blocks) the overhead is 22% and in the worst case when using small messages (16 blocks) the overhead is 45%. The reasons for the increase in overhead as the message count increases relative to the work done, is due mainly to the caching behaviour of the index stream descriptors used on the small 8 KB GPU texture caches. For example, regarding the logical thread index stream when used for a parallel MOO payload, even though it efficiently encodes one logical thread per message (the starting block) and extrapolates the remaining threads for the message, each additional message requires an extra 32 bits. We see a consistent drop off in performance for larger message counts as each binary search performed to map the physical thread ID to message ID must increasingly access global memory. There is also an increased overhead associated with CPU preprocessing of messages and the number of steps involved in the thread to message mapping process. Future work involves attempts to optimise the streams used, even though we are somewhat restricted given that the GPU is an SPMD device and fat kernels add register pressure and reduce occupancy. In particular executing parallel messages in groups of data for large payloads in small message configurations could reduce the overhead of thread to message mapping.

## 7.2 Serial MOOs

The key to good performance with serial MOO messages is to include a lot of messages within the payload. Given a low number of messages, there will be a shortage of threads to maintain a high occupancy level on the GPU and thus performance will suffer. The serial implementations go through the same thread to message mapping process as normal. The message descriptor contains the message size for serial messages, which is used to set the number of input blocks to be processed by each thread starting with the initialisation vector (referenced via the message descriptor). The input address start at the message offset and increase in single blocks treating the message as a contiguous section of the input data stream. This creates a memory access pattern where neighbouring threads access memory locations separated by the size of the messages they are processing. This access pattern has an important impact on throughput as will be

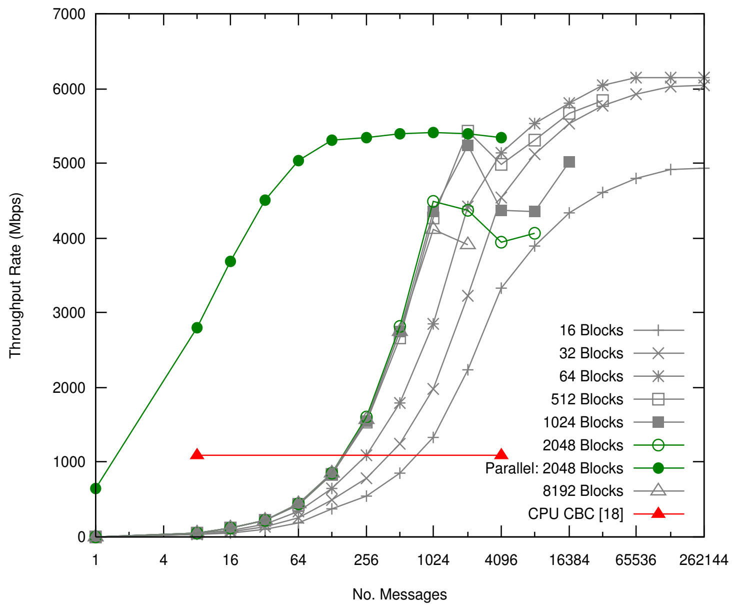


Figure 5: Throughput rates for serial MOO messages across varying block sizes.

seen.

Figure 5 shows the performance rates for a serial MOO using different sizes of messages. All results are based on the CBC MOO in encrypt mode, other serial MOOs using the same block cipher performed equivalently with regards to bulk throughput rates. All messages within a single payload were of the same size, see Section 7.3 for detail on mixing sizes of messages within a payload. We have included in the figure a CPU based implementation of the OFB MOO from [18], as a point of comparison. We have also included the results for a parallel MOO for a payload with a message size of 2048 blocks from Figure 4. The figure highlights its comparison with the corresponding serial MOO message size. We can see the penalty paid for a low number of serial messages within the payload as it takes quite a number of messages before throughput substantially increases. This is easy to see in the comparison of the parallel MOO which starts at quite a reasonable throughput rate from a low message count. We can also see that there is performance to be gained by grouping blocks into threads which reduce the per message overheads discussed above, this accounts for the higher performance of the large message count serial payloads over parallel payloads. These serial results cannot be compared with the AES optimised implementation in Section 4 directly for framework overhead calculations as the optimised AES implementation is implicitly parallel.

A disturbing trend can be observed for larger serial MOO message sizes - as the number of messages increase a performance bottleneck is hit. This may be explained by the memory access pattern created by such executions. Neighbouring threads within a CUDA warp use increasingly disparate memory address locations for their input and output data as the message size increases. We have isolated this behaviour with a separate memory test in which each thread performs a series of sequential reads from global memory starting at an offset from the previous neighbouring thread equal to the number of sequential reads performed. Figure 6 presents these results for different offsets and corresponding sequential reads in increments of blocks (taken to be 16 bytes for this test). For block counts of 128 and over the memory read performance drops dramatically as the the number of active threads increase. There is not enough publicly available detail on the G80 to definitively explain this behaviour, however it is possibly a combination of level 2 cache bottleneck and a limit on the number of separate DRAM open pages supported by the DRAM controllers. Either could cause performance drops as concurrent memory reads reduce their coherency.

## 7.3 Mixed MOOs

Here we investigate the issues involving mixing both the MOOs and message sizes used within a single payload. The same occupancy consideration applies for mixed modes as for single modes, however in a mixed mode context, if a small number of serial MOO messages are present in the payload the presence of parallel MOO messages can help increase occupancy. Performance issues exist when there are imbalances in the number of se-

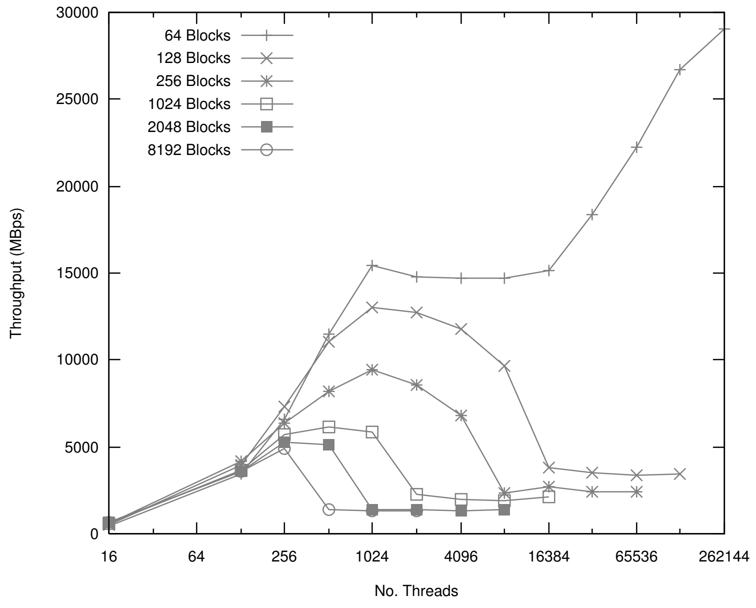


Figure 6: Global memory read performance with varying coherence patterns.

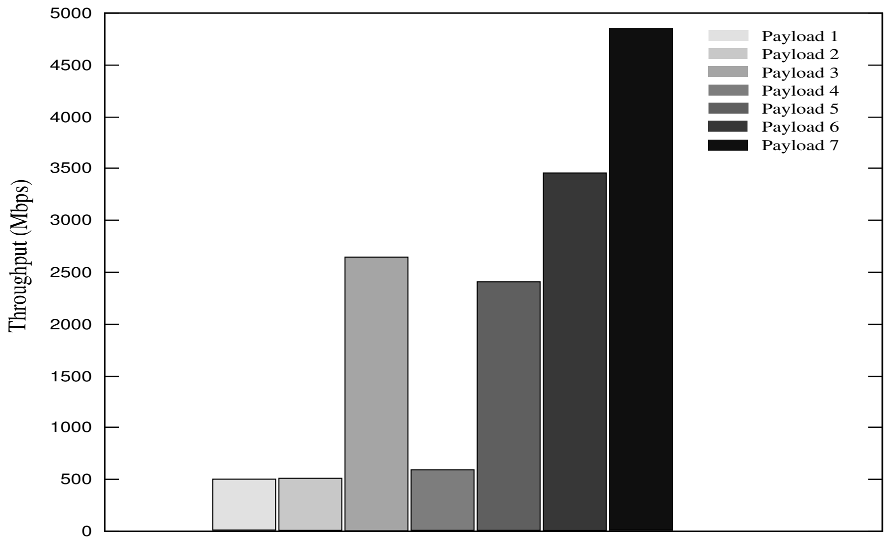


Figure 7: Throughput rates for different payload packing configurations.

rial MOO messages to the amount of parallel MOO message work can be done. Another concern when mixing message types is the positioning of serial MOO messages across the available multiprocessors. The ideal scenario for work load balancing is for all serial messages to be divided evenly across the multiprocessors. Specifically concerning the G80 processor, there is the extra consideration of 32 threads being active at any one time on a single multiprocessor. This restricts the ideal division of serial MOO messages to be made in groups of 32. Also, for optimal arrangement of work to be done within these groups, they should be ordered by message size. This is to reduce the amount of empty SIMD slots during the execution of the serial MOO message groups. Parallel MOO message positioning and message size grouping is not a concern as these types of messages are self load balancing as they are broken up evenly into threads which are load balanced by our thread to message mapping scheme.

To allow the client application sufficient control over the positioning of serial MOO messages on the hardware, we have used the physical thread to message mapping described in Section 6. This allows the client to simply group all serial messages at the start of the payload if possible. The striping mapping scheme used will automatically group the messages into groups of 32 and distribute the groups evenly across CUDA blocks, which will be assigned evenly to the available multiprocessors by the CUDA library. Also, the order in which the serial messages appear in the input stream is preserved, so if the client orders messages according to their size these are maintained in their optimal ordering for SIMD work load balancing. We developed a series of tests which allows us to demonstrate the effect of different mixing configurations of serial MOO messages across a payload. Figure 7 shows the throughput rates of different payload configurations. Each payload configuration consisted of the same messages, only the ordering of the messages were changed. The absolute throughput rates are not relevant as the messages used were manufactured to fit the test requirements and not for performance. The relative difference between the scenarios clearly shows the importance of correct ordering of messages when mixing serial and parallel MOO messages within a single payload.

All payloads used 960 512-block parallel MOO messages, 992 32-block parallel MOO messages and 1024 serial MOO messages with 8 variations in message size ranging from 16 to 2048 blocks. Here is a description of each of the payload configurations used in Figure 7.

**Payload 1:** One serial MOO message per group of 32 threads, attempting to assign all serial MOO messages to a single multiprocessor. This assignment is not entirely possible as CUDA blocks are assigned to available multiprocessors and is beyond the control of the developer.

**Payload 2:** One serial MOO message per 32 threads spread evenly across the multiprocessors.

**Payload 3:** All serial MOO messages assigned to the minimum number of CUDA blocks. This scenario is much faster than 1 and 2 as all SIMD slots within 32 threads are occupied, even though not all multiprocessors are occupied with work.

**Payload 4:** A random distribution of serial MOO messages across the payload.

**Payload 5:** A random distribution of serial MOO messages across the payload, however grouped into 32 threads to ensure full SIMD slot usage.

**Payload 6:** All serial MOO messages grouped into 32 threads and spread evenly across all multiprocessors.

**Payload 7:** Same as Payload 6 however all serial MOO messages appear within the payload in order of their message size. All other payload configurations use a random ordering of message sizes.

From the results one can see the main priorities for client ordering of serial MOO messages within a payload are: their grouping within the device's SIMD width to ensure the SIMD slots are occupied; the even spread of serial MOO message groups across the available mulitprocessors; and the ordering of serial MOO messages according to their size to keep similar message sizes within the one SIMD grouping. A separate and notable concern when mixing function types within a payload is that the underlying implementation can suffer from increased resource pressure. The G80, like other SPMD devices only support a single code path which execute on all threads, thus the combination of function support within a single kernel via conditional blocks can increase register pressure and also increase overhead for the execution of such conditions.

## 8 Conclusion

In this paper we have presented an AES implementation using CTR mode of operation on an Nvidia G80 GPU, which when including data transfer rates shows a 4x speed up over a comparable CPU implementation and a 2x speed up over a comparable GPU implementation. With transfer rates across the PCIe bus not included this ratios increase to 10x and 4x respectively. We have also investigated the use of the GPU for serving as a general purpose private key cryptographic processor. The investigation covers the details of a suitable general purpose data structure for representing client requests and how this data structure can be mapped to underlying GPU implementations. Also covered are the implementation and analysis of both major types of encryption modes of operation, serial and parallel. The paper shows the issues and potentially client preventable caveats when mixing these modes of operation within a single kernel execution.

We show that the use of a generic data structure results in an overhead ranging from 16% to 45%. The main reason for the drop in performance is due to the descriptor data streams becoming too large to fit in the small texture working cache size of the G80. This per thread overhead occurs most acutely within the implementation of parallel MOO payloads with small messages and a high message count. It could be argued that in such a case a client would be better implementing a hardcoded approach if the input data structures are known in advance.

Overall we can see that the GPU is suitable for bulk data encryption and can also be employed in a general manner while still maintaining its performance in many circumstances for both parallel and serial modes of operation messages. Even given the overheads of using a generic data structure for the GPU, the performance is still significantly higher than competing implementations assuming chip occupancy can be maintained. However when small payloads are used the GPUs performance under performs both in the general and hardcoded implementations due to the resource underutilisation and the transfer overheads associated with movement of data across the PCIe bus. Further work is required in optimising the mapping of a generic input data structure to threads to improve the noted overheads. Also investigation of authenticated encryption modes of operation should be included in a similar study.

## References

[1] Nvidia Corporation, "CUDA", http://developer.nvidia.com/object/cuda.html

[2] ATI Corporation, ATI CTM Guide, http://ati.amd.com/companyinfo/researcher/ documents/ATI_CTM_Guide.pdf

[3] AMD Corporation, "Accelerated Proccessing Unit (Sift)", company reference available online at - http://download.amd.com/Corporate/ MarioRivas-Dec2007AMDAnalystDay.pdf

[4] O. ARB, D. Shreiner, M. Woo, J. Neider, T. Davis. OpenGL Programming Guide: The Official Guide to Learning OpenGL. Version 2. 2005.

[5] Microsoft, "Direct X Technology", http://msdn.microsoft.com/directx/

[6] Sarah Tariq, Nvidia Corporation, DirectX10 Effects. Siggraph06, Boston, USA, August 2006.

[7] D. Cook, J. Ioannidis, A. Keromytis and J. Luck, "CryptoGraphics: Secret Key Cryptography Using Graphics Cards". RSA Conference, Cryptographer's Track (CT-RSA), February 2005.

[8] O. Harrison, J. Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units". 9th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007), Vienna, Austria, September 10 - 13, 2007. LNCS, Volume 4727/2007, Pages 209-226.

[9] J. Yang, J. Goodman, "Symmetric Key Cryptography on Modern Graphics Hardware". ASIACRYPT 2007: 249-264

[10] A. Moss, D. Page, N. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware". Cryptography and Coding 2007, pp. 369388

[11] A. Satoh, S. Morioka, K. Takano, S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization". ASIACRYPT 2001, LNCS 2248, pp. 239-254, 2001.

[12] J. Wolkerstorfer, E. Oswald, M. Lamberger, "An ASIC Implementation of the AES Sboxes". RSA Conference 02, San Jose, CA, February 2002.

[13] A.Hodjat, D. Hwang, B. Lai, K. Tiri, I. Verbauwhede, A 3.84 Gbits/s AES crypto coprocessor with modes of operation in a 0.18-um CMOS Technology. Proceedings of the 15th ACM Great Lakes Symposium on VLSI 2005, pages 60–63. April, 2005.

[14] M. McLoone, J. McCanny, "High Performance Single Chip FPGA Rijndael Algorithm Implementations". Workshop on Cryptographic Hardware and Embedded Systems, Paris, 2001.

[15] A. Elbirt, W. Yip, B. Chetwynd, C. Paar, "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists". IEEE Trans. of VLSI Systems, 9.4, pages.545-557, August 2001.

[16] N. Costigan, M. Scott. "Accelerating SSL Using the Vector Processors in IBM's Cell Broadband Engine for Sony's Playstation 3", Cryptology ePrint Archive, Report 2007/061, 2007.

[17] J. Daemen, V. Rijmen, The Rijndael Block Cipher, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/, September 1999.

[18] AMD 64 AES Benchmarks, http://www.cryptopp.com/benchmarks-amd64.html

[19] A. Keromytis, J. Wright, T. de Raadt, ”The Design of the OpenBSD Cryptographic Framework”, In Proceedings of the USENIX Annual Technical Conference, June 2003.

[20] M. Matsui, ”How Far Can We Go on the x64 Processors?”, Fast Software Encryption - FSE 2006, LNCS volume 4047, pages 341-358. Springer-Verlag, 2006.

# An Improved Clock-skew Measurement Technique for Revealing Hidden Services

*Sebastian Zander*
*Centre for Advanced Internet Architectures*
*Swinburne University of Technology*
*Melbourne, Australia*
`http://caia.swin.edu.au/cv/szander/`

*Steven J. Murdoch*
*Computer Laboratory*
*University of Cambridge*
*Cambridge, UK*
`http://www.cl.cam.ac.uk/~sjm217/`

## Abstract

The Tor anonymisation network allows services, such as web servers, to be operated under a pseudonym. In previous work Murdoch described a novel attack to reveal such hidden services by correlating clock skew changes with times of increased load, and hence temperature. Clock skew measurement suffers from two main sources of noise: network jitter and timestamp quantisation error. Depending on the target's clock frequency the quantisation noise can be orders of magnitude larger than the noise caused by typical network jitter. Quantisation noise limits the previous attacks to situations where a high frequency clock is available. It has been hypothesised that by synchronising measurements to the clock ticks, quantisation noise can be reduced. We show how such synchronisation can be achieved and maintained, despite network jitter. Our experiments show that synchronised sampling significantly reduces the quantisation error and the remaining noise only depends on the network jitter (but not clock frequency). Our improved skew estimates are up to two magnitudes more accurate for low-resolution timestamps and up to one magnitude more accurate for high-resolution timestamps, when compared to previous random sampling techniques. The improved accuracy not only allows previous attacks to be executed faster and with less network traffic but also opens the door to previously infeasible attacks on low-resolution clocks, including measuring skew of a HTTP server over the anonymous channel.

## 1 Introduction

The Tor [1] hidden service facility allows pseudonymous service provision, protecting the owners' identity and also resisting selective denial of service attacks. High-profile examples where this feature would have been valuable include blogs whose authors are at risk of legal attack [2]. Other Tor hidden websites host suppressed documents, permit the submission of leaked material, and distribute software written under a pseudonym.

As Tor is an overlay network, servers hosting hidden services are accessible both directly and over the anonymous channel. Traffic patterns through one channel have observable effects on the other, thus allowing a service's pseudonymous identity and IP address to be linked. Murdoch [3] described an attack to reveal hidden services based on remote clock skew measurement.

Here, the attacker induces a load pattern on the victim by frequently accessing the hidden service via the anonymisation network or staying silent. The load changes will cause temperature changes of the victim, which in turn induces deviation of the victim's clock from the true time – clock skew. At the same time, the attacker measures the clock skew of a set of candidate hosts. Viewing induced clock skew as a covert channel the attacker can send a pseudorandom bit sequence to the hidden service and see if it can be recovered from the clock skew measurement of all candidates.

The attacker can measure the target's clock skew by obtaining timestamps from the target's clock and comparing these timestamps against the local clock. In previous research, the clock skew was remotely measured by random sampling of timestamps from the clock. This measurement suffers from two sources of noise: variations in packet delay (jitter) and timestamp quantisation. Network jitter is often small and skewed towards zero, even on long-distance paths, if there is no congestion. Quantisation noise depends on the frequency of the target's clock. Depending on the source of available timestamps, the quantisation noise can be significantly larger than the noise introduced by typical network jitter in the Internet.

To minimise the quantisation error, Murdoch proposed to use synchronised sampling instead of random sampling. Here, the attacker synchronises the timestamp requests with the target's clock ticks, attempting to obtain timestamps immediately after the clock tick, where the quantisation error is smallest. This approach has the potential to reduce the quantisation noise to a small margin, independent of the clock frequency.

Synchronised sampling improves the accuracy of clock-skew estimation, especially for low-resolution timestamps, such as the 1 Hz timestamp of the HTTP protocol. It not only improves the attack proposed by Murdoch, but also opens the door for new clock-skew based attacks which were previously infeasible. Furthermore, our technique could be used to improve the identification of hosts based on their clock skew as proposed by Kohno *et al.* [4] if active measurement is possible.

In this paper we propose an algorithm for synchronised sampling and evaluate it in different scenarios. We show that synchronisation can be achieved, and maintained despite network jitter, for different timestamp sources. Our evaluation results demonstrate that synchronised sampling significantly reduces the quantisation error by up to two orders of magnitude. The greatest improvement is achieved for low-frequency timestamps over low network jitter paths.

The paper is organised as follows. Section 2 introduces the concept of hidden services and describes the threat model and current attacks. Section 3 provides necessary background about remote clock skew estimation. Section 4 describes new attacks possible using synchronised sampling and explains how HTTP timestamps are used for clock skew estimation. Section 5 describes our proposed synchronised sampling technique. In Section 6 we show the improvements of synchronised sampling over random sampling in a number of different scenarios. Section 7 concludes and outlines future work.

## 2 Revealing Hidden Services

In this paper we focus on the Tor network [1], the latest generation from the Onion Router Project [5]. Tor is a popular, deployed system, suitable for experimentation. As of January 2008 there are about 2500 active Tor servers. Our results should also be applicable to other low-latency hidden service designs.

### 2.1 Threat Model

We will assume that the attacker's goal is to link the hidden service pseudonym to the identity of its operator (which in practice can be derived from the server IP address). The attacks we present here do not require control of any Tor node. However, we do assume that our attacker can access hidden services, which means she is running a client connected to a Tor network.

We also assume that our attacker has a reasonably limited number of candidate hosts for the hidden service (say, a few hundred). To mask traffic associated with hidden services, many of their hosts are also publicly advertised Tor nodes, so this scenario is plausible. All of our attack scenarios, with one notable exception, require that the attacker can access the candidate hosts directly (via their IP address). To obtain timestamps, we assume the attacker is able to directly access either the hidden service, or another application running on the target. Again, since many hidden servers are also Tor nodes, it is plausible that at least the Tor application is accessible.

Our attacker cannot observe, inject, delete or modify any network traffic, other than that to or from her own computer.

### 2.2 Existing Attacks

Øverlier and Syverson [6] showed that a hidden service could be rapidly located because of the fact that a Tor hidden server selects nodes at random to build connections. The attacker repeatedly connects to the hidden service, and eventually a node she controls will be the one closest to the hidden server. By correlating input and output traffic, the attacker can discover the server IP address.

Murdoch and Danezis [7] presented an attack where the target visits an attacker controlled website, which induces traffic patterns on the circuit protecting the client. Simultaneously, the attacker probes the latency of all the publicly listed Tor nodes and looks for correlations between the induced pattern and observed latencies. When there is a match, the attacker knows that the node is on the target circuit, and so she can reconstruct the path, although not discover the end node.

Murdoch [3] proposed the most recent attack. The attacker induces an on/off load pattern on the target by frequently accessing the hidden service via the anonymisation network during on-periods, and staying silent during off-periods. At the same time the attacker measures the clock skew changes of the set of candidate hosts. The induced load changes will cause temperature changes on the target, which in turn cause clock skew changes. Viewing the load inducement as covert channel, the attacker can send a pseudorandom bit sequence and compare it with the bit sequences recovered from all candidates through the clock skew measurements. Increasing the duration of the attack increases the accuracy to arbitrary levels.

## 3 Clock-Skew Estimation

All networked devices, such as end hosts, routers, and proxies, have clocks constructed from hardware and software components. A clock consists of a crystal oscillator that ticks at a nominal frequency and a counter that counts the number of ticks. The actual frequency of a device's clock depends on the environment, such as the temperature and humidity, as well as the type of crystal.

It is not possible to directly measure a remote target's true clock skew. However, an attacker can measure the offset between the target's clock and a local clock, and then estimate the relative clock skew. For a packet $i$ including a timestamp of the target's clock received by the measurer, the offset $\tilde{o}_i$ is [3]:

$$\tilde{o}_i = \tilde{t}_i - t_{r_i} = s_c t_{r_i} + \int_0^{t_{r_i}} s(t)dt - c_i/h - d_i \qquad (1)$$

where $\tilde{t}_i$ is the estimated target timestamp (including quantisation error), $t_{r_i}$ is the (local) time the packet was received, $s_c$ is the constant clock-skew component, the integral over *s(t)* is the variable clock skew component, $c_i/h$ is the quantisation noise (for random sampling) and $d_i$ is the network delay.

The constant clock skew is estimated by fitting a line above all points $\tilde{o}_i$ while minimising the distance between each point and the line above it using the linear programming algorithm described in [8]. This leaves the variable part of the clock skew and the noise. To estimate the variable clock skew per time interval, we can use the same linear programming algorithm for each time window $w$.

Figure 1 shows an example of a clock skew measurement across the Internet. The target was 22 hops away with an average Round Trip Time (RTT) of 325 ms. The target was a PC with Intel Celeron 2.6 GHz CPU running FreeBSD 4.10, and measurements were taken from the TCP timestamp clock, which has a frequency of 1 kHz. No additional CPU load was generated on the target during the measurement.

The constant clock skew $s_c$ has already been removed. The grey dots $(\cdot)$ are the offsets between the two clocks, the green line $(-)$ on top is the piece-wise estimation of the variable skew and the blue triangles $(\triangle)$ are the negated values of the derivative of the variable clock skew (the negated clock skew change).

The noise apparent in the figure has two components: the network jitter (on the path from the target to the attacker) and the quantisation error. Note that the network jitter also contains noise inherent in measuring when packets are received by the attacker, and noise caused by variable delay at the target between generating the timestamps and sending packets. In Figure 1, we can clearly see the 1 ms quantisation noise band below the estimated slope, caused by the target's 1 kHz clock. Offsets below this band were also affected by network jitter.

The samples close to the slope on top are the samples obtained immediately after a clock tick (with negligible network jitter). The samples at the bottom of the quantisation noise band are samples obtained immediately before the clock tick. With the linear programming algorithm, only the samples close to the slope on top contribute to the accuracy of the measurement. Assuming an uncongested network, the network jitter is skewed towards zero and small even on long-distance links (see Figure 10). The quantisation noise is inversely proportional to frequency of the target's clock. Depending on the source of timestamps used and the target's operating system, the clock frequency is typically between 1 Hz and 1 kHz (resulting in a noise band between 1 s and 1 ms). If the target does not expose a high-frequency clock, the quantisation noise can be significantly larger than the noise caused by network jitter.

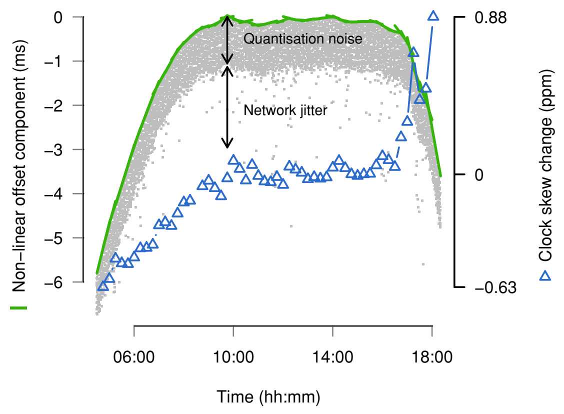


Figure 1: Estimating the variable clock skew

To increase the accuracy of the measurement in the presence of high quantisation noise, $w$ must be set to larger values, as the probability of getting samples close to the slope on top increases with the number of samples. However, large $w$ only allow very coarse measurements. Oversampling provides more fine-grained results while keeping $w$ large to minimise the error. Without oversampling the time windows do not overlap and the start times of windows are $S = \{0, w, 2w, \ldots, nw\}$. With oversampling, the windows overlap and hence the windows start at times $S = \{0, w/o, 2w/o, \ldots, nw/o\}$, where $o$ is the oversample factor.

However, even with large values of $w$, over-sampling has a number of drawbacks. The first estimate is obtained after $w/2$ (regardless of $o$), meaning that for large $w$ it is impossible to get estimates close to the start and end of measurements. Furthermore, large $w$ make it impossible to accurately measure steep clock-skew changes. For example, such changes happen when a CPU load inducement is started and the temperature increases quickly [3]. Another disadvantage of oversampling is the increased computational complexity of $O(o \cdot n \cdot w)$ compared to $O(n \cdot w)$ without over-sampling to obtain the same number of clock skew estimates per time interval.

## 3.1 Timestamp Sources

Previous research used different timestamps sources for clock-skew estimation: ICMP timestamp responses, TCP timestamp header extensions or TCP sequence numbers [3, 4].

TCP sequence numbers on Linux are the sum of a cryptographic result and a 1 MHz clock. They provide good clock-skew estimates over short periods because of the high frequency, but re-keying of the cryptographic function every five minutes makes longer measurements non-trivial [3].

ICMP timestamps have a fixed frequency of 1 kHz. Their disadvantage is that they are affected by clock adjustments done by the Network Time Protocol (NTP) [9], which makes estimation of variable clock skew more difficult. Furthermore, ICMP messages are now blocked by many firewalls.

TCP timestamps have a frequency between 1 Hz and 1 kHz, depending on the operating system. Their advantage is that they are generated before NTP adjustments are made [4]. TCP timestamps are currently the best option for clock-skew measurement because they are widely available and unaffected by NTP (at least for Linux, FreeBSD and Windows [4]). However, even TCP timestamps are not available in all situations. They may not be enabled on certain operating systems and they cannot be used if there is no end-to-end TCP connection to the target. For example, they cannot be used through the Tor anonymisation network.

HTTP timestamps have a frequency of 1 Hz and are available from every web server. However, these have not been previously used for clock-skew measurement due to the low frequency. We describe how to exploit them in the following section.

## 4 New Attacks

A major disadvantage of the attack in [3] is that the attacker needs to exchange large amounts of traffic with the hidden service across the Tor network in order to accurately measure clock skew changes. It may not be possible to actually send sufficient traffic because Tor does not provide enough bandwidth, or because the service operator actively limits the request rate to avoid overload, prevent Denial of Service (DoS) attacks etc. Furthermore, the attack also relies on an exposed high-frequency timestamp source (experiments used the 1 kHz TCP timestamp) on the target for adequate clock-skew estimation.

The synchronised sampling technique proposed in this paper improves the existing attack reducing the duration and amount of network traffic required. Measuring clock-skew requires only a small amount of traffic compared to the amount of traffic needed for load inducement. For example, the exchange of one request/response every 1.5 s is sufficient for clock-skew estimation.

Our improvements also make the existing attack applicable in situations where high-resolution timestamps are not available. For example ICMP or TCP timestamps (see Section 3.1) are not available across Tor, since it only supports TCP and streams are re-assembled on the client, removing any headers. Because our proposed technique allows accurate clock-skew estimation from low-resolution timestamps, HTTP timestamps obtained from a hidden web server across the Tor network could be used. The fact that low-resolution timestamps are usable opens the door to new variants of the attack.

In the first new attack variant, the attacker measures the variable clock skew of the hidden service via Tor, and of all the candidate hosts via accessing the IP addresses directly. Then the attacker compares the variable clock-skew pattern of the hidden service with the patterns of all the candidates. The variable clock skew patterns of different hosts differ sufficiently over time, and the duration of the attack could be increased arbitrarily. While this attack has the benefit of not requiring large amounts of traffic to be exchanged, it could still take a long time. The attacker must ensure that both timestamp sources are derived from the same physical source

A quicker version of this attack could only compare the fixed clock-skew of the target measured via Tor with the fixed clock-skew measured directly for all candidates. Kohno *et al.* showed that clock skew of a particular host changes very little over time, but the difference between different hosts is significant [4].

Another new attack variant is based on the idea of using clock-skew estimates for geo-location [3]. The attacker identifies the location of the candidates based on their IP addresses and a geo-location database. For example, GeoLite is a freely available database that maps IP addresses to locations with a claimed accuracy of over 98% [10]. The attacker measures the variable clock skew of the hidden service via Tor. The attacker then estimates the location based on the variable clock-skew pattern using the technique described in [3].

This attack works even in cases where the attacker cannot access the candidate hosts directly. On the other hand this attack does not allow an unambiguous identification of the hidden service if candidate locations are geographically close together.

## 4.1 Attacking HTTP Timestamps

The key common factor among the new attacks discussed above is that clock skew must be estimated from responses sent over the hidden service channel. Previous work has not examined this option because typically

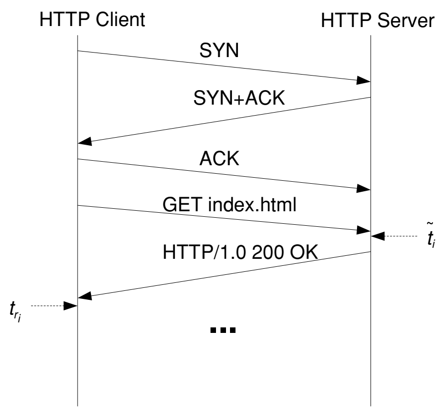


Figure 2: HTTP request/response and the timestamps needed for clock skew estimation

only a low frequency clock is available and quantisation noise dominates the small effect of temperature on clock skew. However, in this paper we will show how it is still possible to exploit this clock. Here, the attacker acts as HTTP client sending minimal HTTP requests to the target. Standard web servers include a 1 Hz timestamp in the Date header of HTTP responses because it was recommended for HTTP 1.0 [11] and is mandatory for HTTP 1.1 (excluding 5xx server errors and some 1xx responses) [12].

Before the HTTP exchange a TCP connection needs to be established between client and server. Including TCP connection establishment, it may take at least two full round trip times from when the client wants to send a request until the response is received. To minimise the TCP overhead, the client should open a TCP connection beforehand. Ideally, it should only open the connection once and then keep it alive for the duration of the measurement. However, it is not possible for a client to force a server to keep a connection open. Therefore, when the client notices that the server has closed the connection, it should immediately re-open it. Then, the next HTTP request can be sent at the appropriate time determined by the synchronised sampling algorithm.

The HTTP timestamp is usually generated after the server has received the client's request. We verified that Apache 2.2.x generates the timestamp after the request has been successfully parsed [13]. The corresponding client timestamp is the time the packet containing the Date header is received, which is usually the first packet sent by the server after the TCP connection has been fully established (see Figure 2).

## 5 Implementation

Previous approaches to remote clock-skew estimation have sampled timestamps at random times. However, with the 1 Hz HTTP timestamp, the consequent high quantisation noise would prevent the attack from accurately measuring the target's clock skew within a feasible period. Instead, we must probe the target's clock immediately after a clock tick occurred, because here the quantisation error is the smallest. To achieve this, the attacker has to synchronise its probing phase and frequency such that probes arrive shortly after the clock tick. We assume the attacker selects a nominal sample frequency, based on the desired accuracy and intrusiveness of the measurement.

The attacker cannot measure the exact time difference between the arrival of probe packets and the clock ticks. To maintain synchronisation, the attacker has to alternate between sampling before and after the clock tick. Samples before the clock tick can be corrected by adding one tick, as their true value is actually closer to the next clock tick. However, the linear programming algorithm still cannot use these samples because for them the quantisation error and jitter are in opposite directions and cannot be separated.

Figure 3 illustrates the benefit of synchronised sampling over random sampling. The solid step line is the target's clock value over time and the dashed line shows the true time. Random samples are distributed uniformly between clock ticks whereas synchronised samples are taken close to the clock ticks. Note that in the figure, the time for samples before the tick has been corrected as described above. The quantisation errors are the differences between samples' $y$-values and the true time. The absolute quantisation errors are shown as bars at the bottom. Synchronised sampling leads to smaller errors in comparison with random sampling.

Our algorithm is similar to existing Phase Lock Loop (PLL) techniques used for aligning the frequency and phase of a signal with a reference [14]. However, whereas PLL techniques measured the phase difference of two signals, we can only estimate the phase difference by detecting whether a sample was taken before or after the clock tick.

### 5.1 Algorithm

Initially, the attacker starts probing with the nominal sample frequency and measures how many clock ticks occur in one sample interval (*target_ticks_per_interval*). The measurement is repeated to obtain the correct number of ticks.

The attacker cannot measure the exact time difference between the arrival of a probe packet and the target's clock tick. However, the attacker can measure the position of the probe arrival relative to the target's clock tick based on the number of clock ticks that occurred between the current and the last timestamp of the target (*ticks_diff*). If the number of clock ticks is less

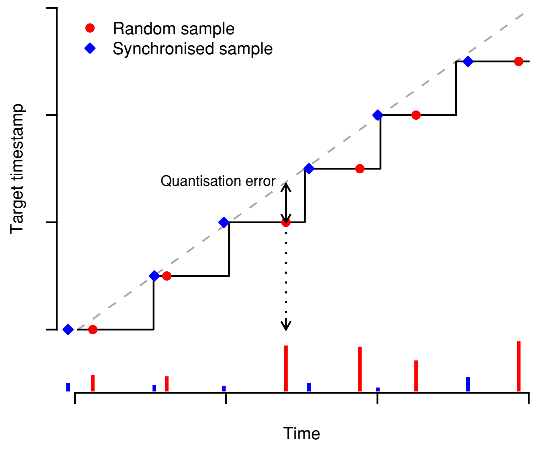


Figure 3: Advantage of synchronised sampling over random sampling

than *target_ticks_per_interval*, the sample was taken before the tick and vice versa. If *ticks_diff* equals *target_ticks_per_interval* the position is left unchanged. At the start of the measurement the position is not known and the attacker needs to continuously increase or decrease the probe interval until a change occurs (initial phase lock).

The probe interval (the reciprocal of the probe frequency) is controlled using the following mechanism (see Algorithm 1). The probe interval is adjusted based on the position errors each time a position change occurs and the previous position was known using a Proportional Integral Derivative (PID) controller [15]. PID controllers base the adjustment not only on the last error value, but also on the magnitude and the duration of the error (integral part) as well as the slope of the error over time (derivative part). $K_p$, $K_i$ and $K_d$ are pre-defined constants of the PID controller.

Alternatively, the linear programming algorithm [8] could be used to compute the relative clock skew between attacker and target based on a sliding window of timestamps. The probe interval is then adjusted based on the estimated relative clock skew. This technique works well if the estimates are fairly accurate, which is the case for high-frequency clocks and low network jitter.

**Algorithm 1** Probe interval control

```
function probe_interval_adjustment(pos, last_pos)
  if pos != last_pos and pos != UNKNOWN and
     last_pos != UNKNOWN then
    return $K_p$·(last_adj_before + last_adj_behind) +
           $K_i$·(integ_adj_before + integ_adj_behind) +
           $K_d$·(deriv_adj_before + deriv_adj_behind)
  else
    return 0
```

In order to maintain the synchronisation, the attacker has to enforce regular position changes. This is done by modifying the time the next probe is sent. If the current position is before the clock tick the send time of the next probe is increased based on the last adjustment *last_adj_before*. If the current position is behind the clock tick the next probe send time is decreased based on the last adjustment *last_adj_behind*. The adjustments are modified based on how well the attacker is synchronised to the target.

If a change of position occurs between two samples, the difference between the arrival of the probe packet and the target's clock tick is smaller than the last adjustment and therefore the next adjustment is decreased. If no position change occurs the error is assumed to be larger than the last adjustment and the next adjustment is increased. The initial probe send-time adjustment is a pre-defined constant. Algorithm 2 shows the probe send time adjustment algorithm. $\alpha$ and $\beta$ are pre-defined constants that determine how quickly the algorithm reacts ($0 < \alpha < 1$ and $\beta > 1$).

**Algorithm 2** Next probe send time adjustment

```
function next_probe_time_adjustment(pos, last_pos)
  if pos = BEFORE then
    last_adj = last_adj_before
  else
    last_adj = last_adj_behind

  if pos != last_pos then
    return $\alpha$·last_adj
  else
    return $\beta$·last_adj
```

The probe frequency and send time adjustments are limited to a range between pre-defined minimum and maximum values to avoid very small or very large changes.

Loss of responses is detected using sequence numbers. A sequence number is embedded into each probe packet such that the target will return the sequence number in the corresponding response. The actual field depends on the protocol used for probing. For example, for ICMP the sequence number is the ICMP Identification field whereas for TCP the sequence number is the TCP sequence number field.

For HTTP it is not possible to embed a sequence number directly into the protocol. Instead, sequence numbers are realised by making requests cycling through a set of URLs. A sequence number is associated with each URL and HTTP responses are mapped to HTTP requests using the content length assumed to be known for each object. This technique assumes there are multiple objects with different content lengths accessible on the web server.

If packet loss is detected, the algorithm adjusts *ticks_diff* by subtracting the number of lost packet multiplied by *target_ticks_per_interval*. Reordered packets are considered lost.

Algorithm 3 shows the synchronisation procedure. Our algorithm works with different timestamps and different clock frequencies. It has been tested with ICMP, TCP and HTTP timestamps and TCP clock frequencies of 100 Hz, 250 Hz and 1 kHz.

**Algorithm 3** Synchronised sampling

```
foreach response_packet do
  diff = target_timestamp - last_target_timestamp

  if target_ticks_per_interval == -1 then
    pos = UNKOWN
    target_ticks_per_interval = ticks_diff
  else if ticks_diff > target_ticks_per_interval then
    pos = BEHIND
  else if ticks_diff < target_ticks_per_interval then
    pos = BEFORE
  else
    pos = last_pos

  probe_interval = probe_interval +
             probe_interval_adjustment(pos, last_pos)
  probe_time = last_probe_time + probe_interval +
            next_probe_time_adjustment(pos, last_pos)

  last_pos = pos
  last_target_timestamp = target_timestamp
  last_probe_time = probe_time
```

## 5.2 Errors

Any constant delay does not affect the synchronisation process. However, changes in delay on the path from the attacker to the target will affect the arrival time of the probe packets. This could be caused by jitter in the sending process (jitter inside the attacker), network jitter (queuing delays in routers) or jitter in the target's packet receiving process.

Often we can assume the network is uncongested and therefore network jitter is skewed towards zero. This is usually the case in a LAN (see Section 6). Even on the Internet many links are not heavily utilised and path changes (caused by routing changes) are usually infrequent. Load-balancing is usually performed on a per-flow basis to eliminate any negative impacts on TCP and UDP performance.

However, when measuring clock skew over a Tor circuit we expect much higher network jitter. A Tor circuit is composed of a number of network connections between different nodes. The overall jitter does not only include the jitter of each connection but also the jitter introduced by the Tor nodes themselves.

The timing of sending probes is not very exact if the sender is a userspace application. Even if the userspace *send()* system call is called at the appropriate time, there will be a delay before the packet is actually sent onto the physical medium. The variable part of this delay can cause probe packets to arrive too late or too early. This error could be reduced by running the software entirely in the kernel (e.g. as kernel module), using a real-time operating system or by using special network cards supporting very precise sending of packets. Any variable delay in the packet receiving process of the target has the same effect and is unfortunately out of control of the attacker. The only way an attacker could reduce such errors would be to adjust the sending of the probe packets based on a prediction of the jitter inside the target, which appears to be a challenging task.

Another error is introduced when the relative clock skew between attacker and target changes and the algorithms needs to adjust the probe frequency. The attacker is able to control its time keeping and avoid any sudden clock changes. But if the target is running NTP and the timestamps are affected by NTP adjustments, changes in relative clock skew are possible.

## 6 Evaluation

In the first part of this section we compare the accuracy of synchronised and random sampling in a LAN testbed using TCP timestamps with typical target clock frequencies of 100 Hz, 250 Hz and 1000 Hz, as well as 1 Hz HTTP timestamps. Since in the LAN network jitter is negligible the results show the maximum improvement of using synchronised sampling and demonstrate that our implementation is working correctly.

In the second part we compare the accuracy of synchronised and random sampling based on TCP timestamps across a 22-hop Internet path. The result shows that even on a long path, synchronised sampling significantly increases clock-skew estimation accuracy, which improves on the attack proposed by Murdoch [3].

In the third part we compare the accuracy of synchronised and random sampling for probing a web server running as a Tor hidden service. We show that synchronised sampling improves clock skew estimation significantly, even over a path with high network jitter. We also show that a hidden web sever can be identified among a candidate set by comparing the variable clock skew over time using synchronised sampling. Furthermore, using synchronised sampling shows daily temperature patterns that could not be identified using random sampling.

Finally, we investigate how long our technique needs for the initial synchronisation (the time until the attacker has locked on to the phase and frequency of the target's clock ticks). We compare the times for HTTP probing in a LAN and probing a hidden web server over a Tor network.

To evaluate the accuracy of synchronised and random sampling we need to know the true values of the variable clock skew. Since it is impossible to directly measure this, we use the following approach. In our tests the target also runs a UDP server and the attacker runs a UDP client. The UDP client sends requests to the server at regular time intervals. Upon receiving a request, the UDP server returns a packet with a timestamp set to the send time of the response. The UDP client records the time it receives the response.

We compute the offset between the two timestamp-series and estimate the variable clock skew as usual. Since the UDP-based timestamp has a precision of 1 µs, the quantisation error is negligible. Although these UDP estimates are not the true values of the variable clock skew we use them as baseline for synchronised and random sampling, which have much higher quantisation errors. In the following we refer to this as UDP probing or UDP measurement.

A drawback of our current implementation is that the UDP server is a userspace program. The server's response timestamp is taken in userspace before the response packet is sent via the *sendto()* system call. To reduce these timing errors one could implement a kernel-based version of the UDP server.

We compare the variable skew estimates for synchronised and random sampling with the reference values, from the UDP measurement, using the root mean square error (RMSE) of the data values $x$ against the reference values $\hat{x}$:

$$RMSE = \sqrt{\frac{1}{N}\sum_i (\hat{x}_i - x_i)^2}. \tag{2}$$

We also compute histograms of the noise band for synchronised and random sampling. The noise is defined as difference between the variable clock offset and the UDP timestamp estimated variable skew. For random sampling the quantisation noise band is always uniform with width $1/f$, where $f$ is the clock frequency. For synchronised sampling the quantisation noise depends on how well the synchronised sampling algorithm is able to track the target's clock tick. For synchronised sampling the noise is given by the samples taken after the clock tick because only these samples are used to estimate the clock skew.

In all experiments we set $\alpha = 0.5$ and $\beta = 1.5$. For TCP timestamps the linear programming algorithm was used to adjust the probe interval with a sliding window of size 120 (LAN) and 300 (Internet). For HTTP timestamp measurements the probe interval was adjusted using the PID controller with $K_p = 0.09$, $K_i = 0.0026$ and $K_d = 0.02$.

## 6.1 Synchronised vs. Random Sampling in LAN Environments

The attacker was a PC with Intel Xeon 3.6 GHz Quad-Core CPU running Linux 2.6. The target was a PC with Intel Xeon 3.6 GHz Quad-Core CPU running Linux 2.6 with a TCP timestamp frequency of 1 kHz. Attacker and target were connected to the same Ethernet switch. The attacker simultaneously performed synchronised, random and UDP probing. Synchronised and random probing had an average sampling period of 1.5 s, the same rate as in [3]. The UDP probing was performed with a faster sample rate of 1 s in order to achieve a higher accuracy for the reference measurement. A second UDP measurement with an average sample rate of 1 s was run in order to investigate the error between UDP measurements. The duration of the test was approximately 24 hours.

As the test was run inside a LAN the average RTT was only 130 µs and the RTT / 2 jitter was small with a maximum of 60 µs and a median of 30 µs. Figure 4 shows histograms of the noise bands of synchronised and random sampling with respect to the reference given by the UDP measurement. For synchronised sampling most of the offsets are within 100 µs whereas for random sampling we see the expected 1 ms noise band.

In Figure 5 we compare the RMSE of synchronised sampling, random sampling and the second UDP measurement for different window sizes against the UDP reference with maximum window size (1800 s). We also compare the UDP reference against itself at smaller window sizes. The oversampling factor was chosen such that the time between two clock-skew estimates is the same regardless of the window size (30 s). This has the advantage of providing approximately the same number of estimates for all window sizes. (For smaller window sizes there are still more samples, because there are samples closer to the start and end of the measurement period.)

Figure 5 shows that synchronised sampling performs significantly better than random sampling. There is a difference between the second UDP measurement and the UDP reference, but it is smaller than the difference between synchronised sampling and the UDP reference for all window sizes. Hence we conclude the error of UDP measurements is sufficiently small for using it as baseline. In the later experiments we performed only one UDP measurement.

The target clock frequency was 1 kHz, which is the maximum TCP timestamp frequency of current operating systems. However, it is likely that in reality many hosts actually have lower TCP clock frequencies. For example, 100 Hz is the clock frequency used by older Linux

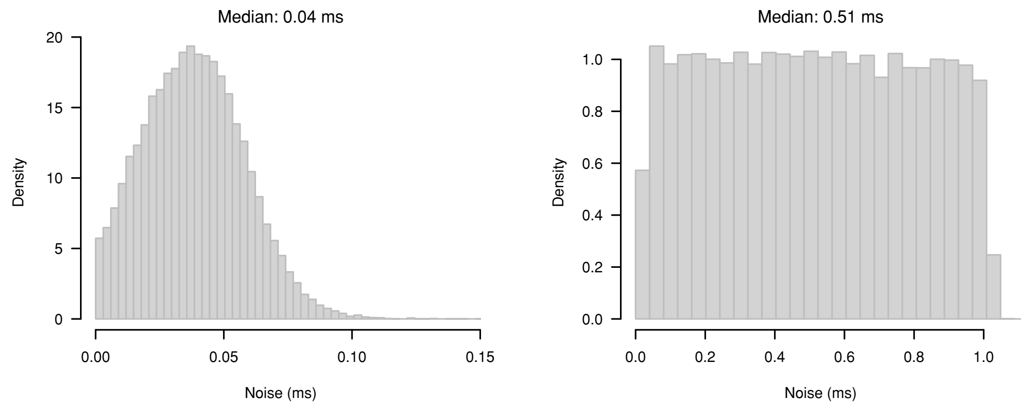


Figure 4: Noise distributions in LAN: synchronised sampling (left) vs. random sampling (right)

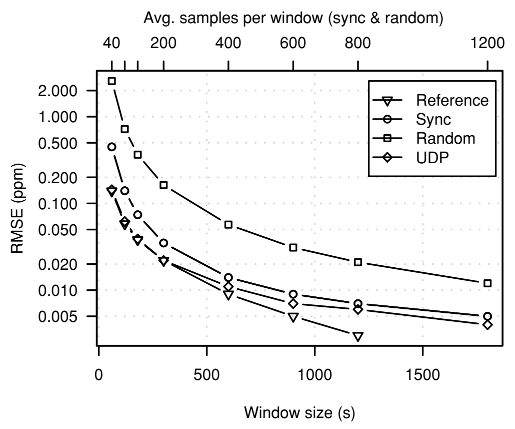


Figure 5: RMSE of synchronised, random sampling and UDP reference in LAN with a target clock frequency of 1 kHz (log *y*-axis)

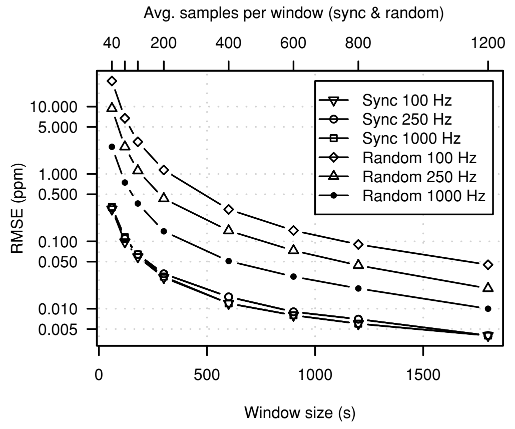


Figure 6: RMSE of synchronised and random sampling for different clock frequencies of 100 Hz, 250 Hz and 1 kHz against the same target. Different clock frequencies were obtained by rounding the target timestamps immediately after reception (log *y*-axis)

and FreeBSD kernels and 250 Hz is the clock frequency of modern Linux 2.6 kernels.

To evaluate the RMSE for lower clock frequencies we used the same setup. This time we ran three synchronised and three random probing processes simultaneously for 24 hours, rounding the target timestamps so that we effectively measured 100 Hz, 250 Hz and 1 kHz clocks. Figure 6 shows the RMSE for synchronised and random sampling for the different target clock frequencies. The UDP measurement has been omitted for better readability. The graph shows that the accuracy of synchronised sampling does not depend on the clock frequency and the RMSE for random sampling increases significantly for lower clock frequencies.

In another LAN experiment we ran a web server (Apache 2.2.4) on the target and the attacker used HTTP probing. The average sampling interval was 2 s, because this is the minimum probe frequency for 1 Hz HTTP timestamps. The web server was completely idle (except for the requests generated by the attacker). The duration of the experiment was approximately 24 hours.

Although the experiment was carried out between the same two hosts as before, the RTT / 2 jitter was higher with a maximum of 120 µs and a median of 60 µs. The web server running in userspace introduced the additional jitter. Figure 7 shows the noise for synchronised sampling and random sampling. For synchronised sampling the noise band is only slightly larger than in Figure 4. Because of the higher jitter, the synchronisation is less accurate. For random sampling the noise band is 1 s because of the 1 Hz HTTP clock frequency.

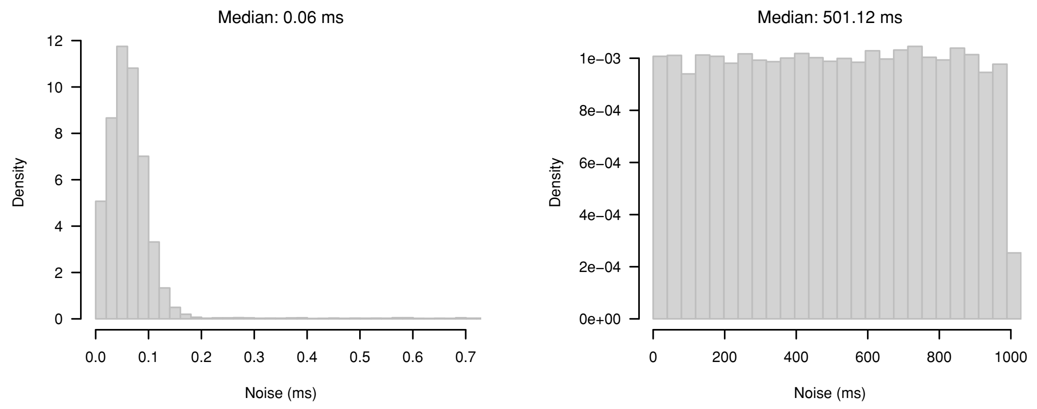


Figure 7: Noise distributions for HTTP probing in LAN: synchronised sampling (left) vs. random sampling (right)

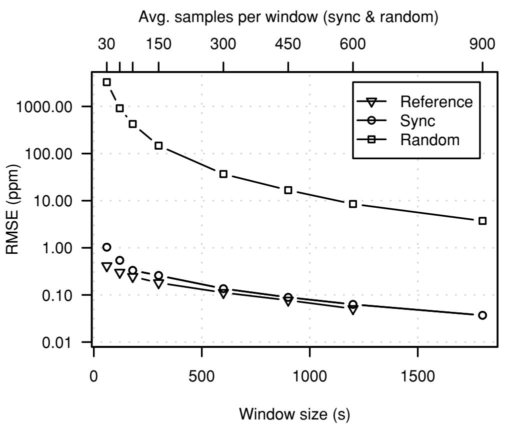


Figure 8: RMSE of synchronised sampling, random sampling and the UDP measurement for HTTP probing in LAN

Figure 8 shows the RMSE of synchronised sampling, random sampling and the UDP measurement against the reference at maximum window size. The RMSEs of synchronised sampling and UDP reference are very similar to the results in Figure 5. Because of the large noise band, the RMSE for random sampling is more than two orders of magnitude above the RMSE for synchronised sampling. This demonstrates that our new algorithm is able to effectively measure clock skew changes for low frequency clocks, an infeasible task for random sampling.

## 6.2 Synchronised vs. Random Sampling Across Internet Paths

The attacker was the same machine as in Section 6.1 located in Cambridge, UK. The target was 22 hops away located in Canberra, Australia. The target was a FreeBSD 4.10 PC with a kernel tick rate set to 1000 and therefore the TCP timestamp frequency was 1 kHz. The average RTT between measurer and target was 325 ms. The duration of the measurement was approximately 21 hours. We performed synchronised, random and UDP probing.

Despite the high RTT, the jitter is small and skewed towards zero as shown in Figure 10. Figure 9 shows histograms of the noise bands of synchronised and random sampling in relation to the reference given by the UDP measurement. For synchronised sampling most of the offsets are within 250 µs of the reference whereas for random sampling there is the expected 1 ms noise band.

Figure 11 shows the RMSE of synchronised sampling, random sampling and the UDP reference against the UDP reference at maximum window size using the same parameters as in Section 6.1. The gain of synchronised sampling is smaller compared to Section 6.1 because of the higher network jitter but still significant for smaller window sizes.

## 6.3 Attacking Tor Hidden Services

For our measurements we used a private Tor network. Our Tor nodes are distributed across the Internet running on Planetlab [16] nodes. The main reason for using a private Tor network instead of the public Tor network is the poor performance of hidden services in the public Tor network. Besides huge network jitter that prevents any accurate clock-skew measurements, hidden services always disappeared after few hours preventing longer measurements. While currently it is difficult to carry out the attack in the public Tor network, it should become easier in the future, as the Tor team is now working on improving the performance of hidden services.

We selected 18 Planetlab widely geographically distributed nodes on which we ran Tor nodes (of which 3 were directory authorities). We selected nodes that had low CPU utilisation at the time of selection. An Intel Core2 2.4 GHz with 4 GB RAM running Linux

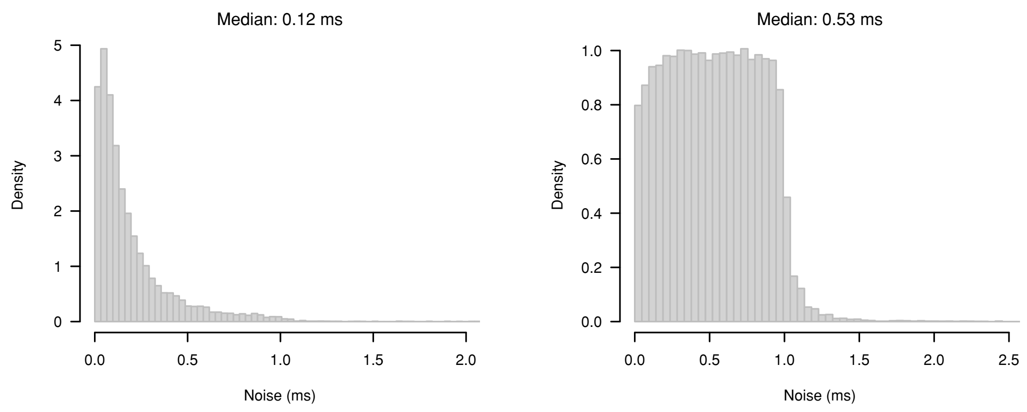


Figure 9: Noise distributions for Internet path: synchronised sampling (left) vs. random sampling (right)

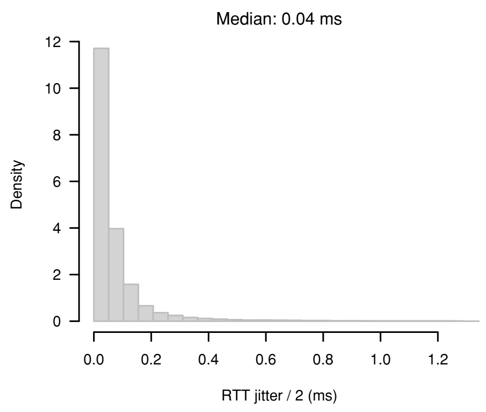


Figure 10: RTT jitter / 2 on path across the Internet

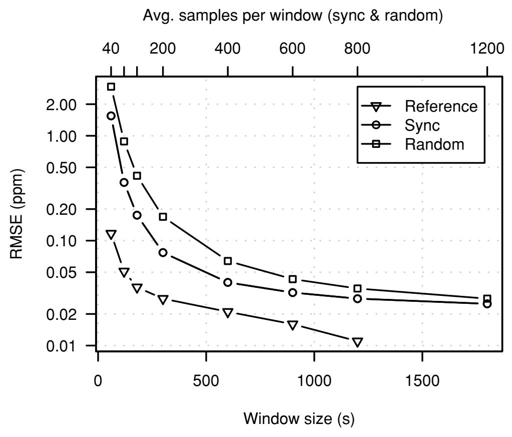


Figure 11: RMSE of synchronised sampling and random sampling for different window sizes measured across Internet path.

2.6.16 was used to run another Tor node and the hidden web server. No load is induced on the server, so any clock skew changes are based on the ambient temperature changes.

An Intel Celeron 2.4 GHz with 1.2 GB of RAM running Linux 2.6.16 was used to run a Tor client and our probe tool. We used tsocks [17] with the latest Tor related patches to enable our tool to interact with the Tor client via the SOCKS protocol and to properly handle Tor hidden server pseudonyms.

First we performed an experiment similar to the ones in Section 6.1 and Section 6.2. Synchronised and random sampling was performed across the Tor network, while UDP probing was performed directly between the client machine and the hidden server. The measurement duration was approximately 18 hours.

The average RTT between client and hidden server across Tor was 885 ms. Figure 14 shows the RTT / 2 jitter, which is considerably higher than in the previous measurements. Figure 12 shows histograms of the noise bands of synchronised and random sampling. For random sampling it shows the expected 1 s noise band. For synchronised sampling the noise is greatly reduced. Most of the offsets are $\leq$ 100 ms away from the slope given by the UDP reference.

Figure 13 shows the change of clock skew for synchronised sampling as blue squares (□) and random sampling as red circles (○) and the UDP reference as black line (−) for a window size of 1800 s and 2 hours. The noise is much smaller for synchronised sampling compared to random sampling especially for small window sizes. For a window size of 2 hours one can clearly see a daily temperature change of the reference curve with the temperature (and hence the clock skew) dropping during night hours and suddenly increasing in the morning. The synchronised sampling curve shows the same pattern with

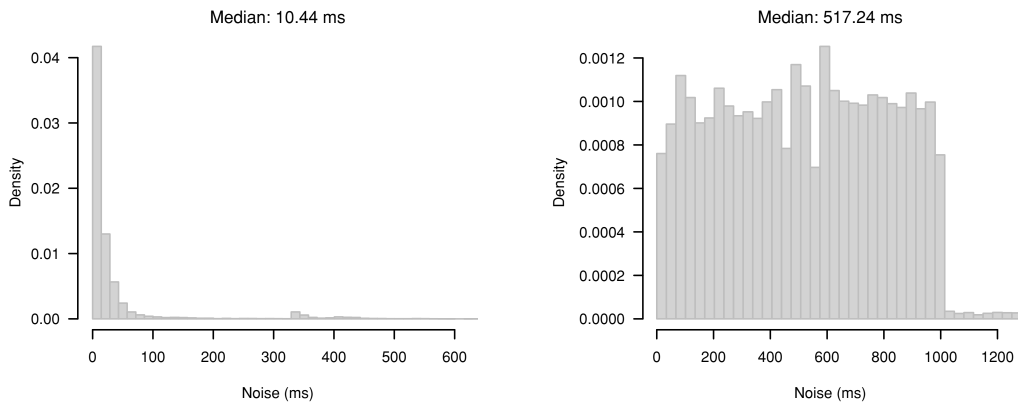


Figure 12: Noise distributions Planetlab Tor testbed: synchronised sampling (left) vs. random sampling (right)

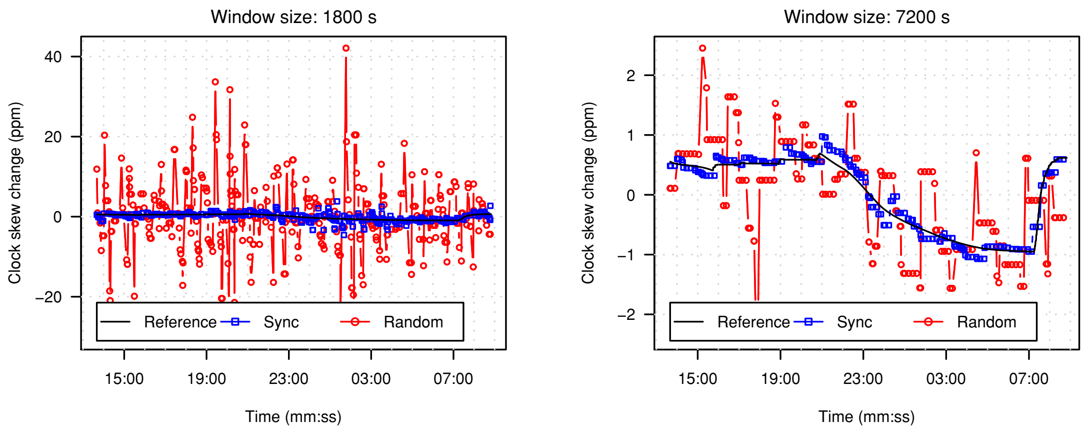


Figure 13: Estimated clock skew changes for hidden service in Planetlab Tor network for a window size of 1800 s (left) and 2 hours (right)

added noise. An attacker could use such daily temperature patterns to estimate the location of the target based on geo-location. In contrast to random sampling, the pattern is not clearly visible because of the much higher noise.

In Figure 15 we compare the RMSE of synchronised sampling, random sampling and the UDP reference against the UDP reference at maximum window size. The RMSE of synchronised sampling is almost one magnitude lower than the RMSE for random sampling even for window sizes as large as two hours.

In the second experiment we performed the actual attack. We treated all 19 Tor nodes as candidates and measured their clock skew directly using TCP timestamps (synchronised sampling). At the same time we measured the clock skew of the hidden web service via Tor based on HTTP timestamps using synchronised and random sampling simultaneously. The experiment lasted about ten hours. One of the nodes stopped responding in the middle of the experiment.

Figure 16 shows the RMSE of the HTTP clock skew estimates obtained from the hidden service via Tor using random sampling or synchronised sampling and TCP clock skew estimates of all candidate nodes. We used a window size of three hours and set the oversample factor so one clock estimate is obtained every 30 s. (For smaller windows random sampling was not able to consistently select one candidate as the best and would alternate between a few including the correct one for the whole duration of the measurement.)

The RMSE of the HTTP timestamp estimate and the correct candidate is shown as thick grey (red on colour display) line while RMSEs for all other candidates are shown as thin black lines.

The RMSE between the synchronised sampling Tor measurement and the direct measurement of the correct candidate is very small, and with increasing duration be-

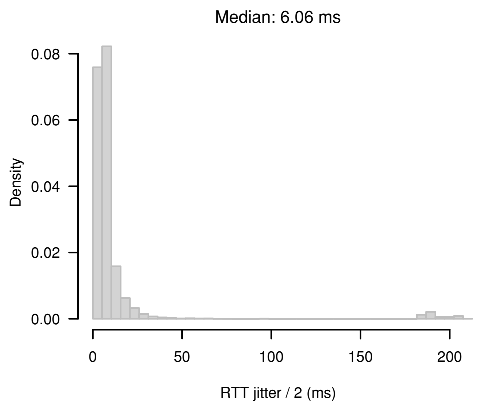


Figure 14: RTT jitter / 2 over Planetlab Tor testbed

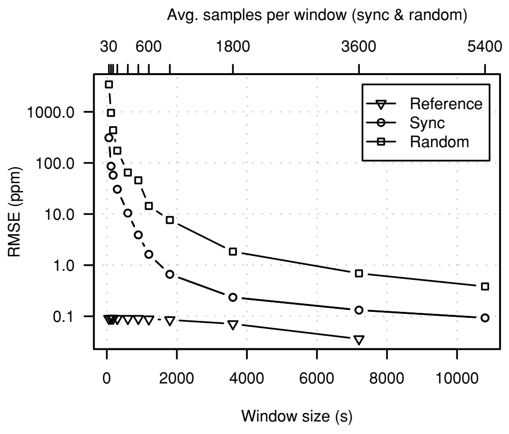
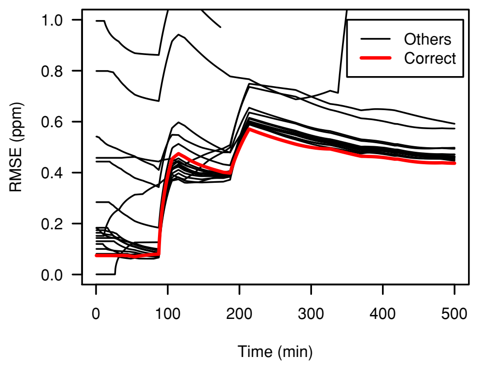


Figure 15: RMSE of synchronised, random sampling and UDP reference for hidden web service in Planetlab Tor network (log *y*-axis)



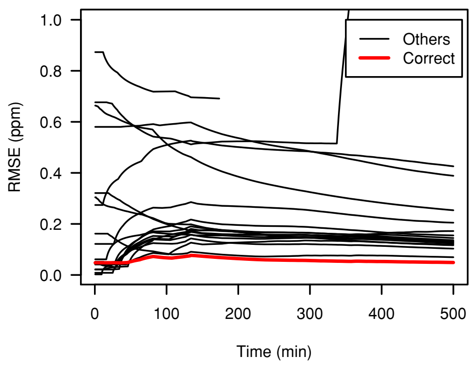


Figure 16: RMSE of HTTP clock skew estimates obtained from hidden service via Tor using random sampling (left) or synchronised sampling (right) and TCP clock skew estimates of all candidate nodes

comes significantly smaller than the RMSE of the Tor measurement and all the other candidates except one. For random sampling all RMSEs are fairly high indicating that there is no good match of the variable clock skew of the Tor hidden service with any of the candidates. In the second half of the experiment the RMSE of the correct candidate becomes smallest, but only by a very small margin.

Synchronised sampling is able to identify the correct candidate much faster than random sampling, needing only 139 minutes compared to 287 minutes. These times are from the start of the measurement until the RMSE of the correct candidate becomes smallest. They include the initial 1.5 hours it takes to get the first clock skew estimate (because of the three hour windows), which is not included in Figure 16.

While the variable clock skew of the TCP clock and userspace clock (HTTP timestamps) are a good match the fixed skew of the two clocks differs on our Linux 2.6.16 box running the hidden server. This makes it impossible to evaluate an identification of the hidden server based on the fixed skew. However, since we know the true fixed skew of the userspace clock, we can analyse how long it takes to get an estimate using synchronised and random sampling of the HTTP clock. We use the data from the previous measurement and assume the skew estimate is correct if within 0.5 parts per million of the true value. Again synchronised sampling outperforms random sampling, needing only 23 minutes compared to 102 minutes.

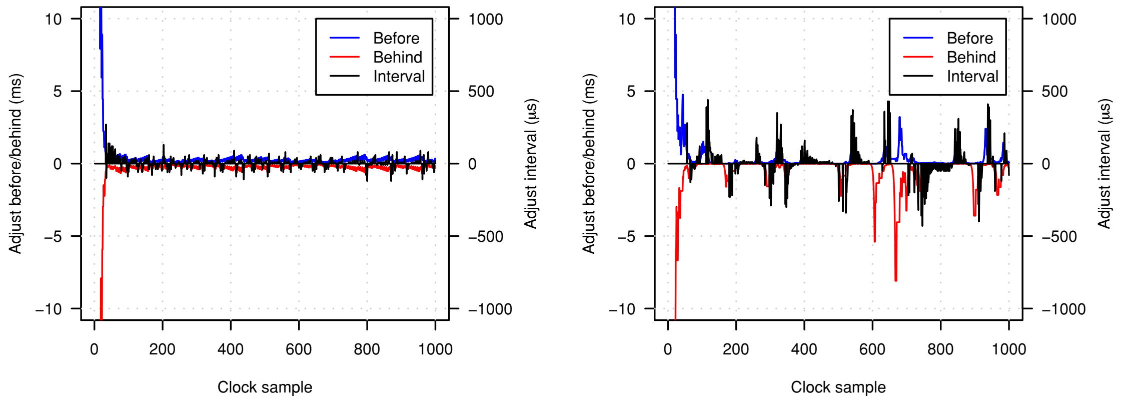


Figure 17: Initial synchronisation for HTTP probing in LAN (left) and probing a hidden web server over the Tor network (right)

## 6.4 Initial synchronisation time

We briefly analyse the initial synchronisation time of our technique. The initial synchronisation is the time it takes until the attacker has locked on to the phase and frequency of the target's clock ticks.

Figure 17 plots the values of *adj_before*, *adj_behind* and *probe_interval* (see Section 5 for the meaning of the variables) over the number of clock samples (taken from the target's clock every 2 s). The $y$-axis range is limited to between $-10$ ms and 10 ms and the $x$-axis is limited to the first 1000 clock samples. Note that before adjustments are always positive, while behind adjustments are always negative.

In the LAN experiment initial synchronisation is established after only about 40 samples (roughly 1.5 minutes) and further adjustments and probe interval changes are small (less than 500 µs and 100 µs respectively). When probing over the Tor network synchronisation is more difficult because of the much higher network jitter. Consequently initial synchronisation takes longer (about 70 clock samples or roughly 2.5 minutes) and the algorithm is forced to make larger adjustments and probe interval changes (of up to several milliseconds).

## 7 Conclusions and Future Work

In this paper we have presented and evaluated an improved technique for remote clock-skew estimation based on the idea of synchronised sampling proposed by Murdoch [3]. The evaluation shows that our new algorithm provides more accurate clock skew estimates than the previous random sampling based approach. Especially if the target clock frequency is low, accuracy improves by up to two orders of magnitude. Since the accuracy of our synchronised sampling technique is independent of the target's clock frequency, it is possible to estimate variable clock-skew from low-resolution timestamps.

Our technique does not only improve the previously proposed clock-skew related attack on Tor [3], it also opens the door for new variants of the attack, which we have described in the paper. Our technique could also be used to improve the identification of hosts based on their clock skew as proposed in [4] if active measurement is possible.

Currently our Tor test network is fairly small and only has one hidden server. While we showed that our new proposed attacks work in principle, we did not provide a comprehensive evaluation. In future work we plan to extend our test network and add more hidden servers. This will allow us to perform a more detailed evaluation including analysing the sensitivity and specificity of our attack based on the different parameters.

The synchronised sampling implementation could be further improved by fine-tuning the algorithm parameters. Our current implementation runs in userspace, which naturally limits the ability to exactly time probe packets. A kernel implementation, using network cards capable of high-precision traffic generation, or use of a real-time kernel, could achieve higher accuracy.

## Acknowledgements

We thank Markus Kuhn and the anonymous reviewers for their valuable comments.

## References

[1] P. Syverson R. Dingledine, N. Mathewson. Tor: The second-generation onion router. In *Proceed-*

*ings of the 13th USENIX Security Symposium*, August 2004.

[2] Reporters Without Borders. Blogger and documentary filmmaker held for the past month, March 2006. `http://www.rsf.org/article.php3?id_article=16810`.

[3] S. J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In *CCS '06: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 27–36, Alexandria, VA, US, October 2006. ACM Press.

[4] T. Kohno, A. Broido, and kc claffy. Remote physical device fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 211–225, May 2005.

[5] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, February 1999.

[6] Lasse Øverlier and Paul F. Syverson. Locating hidden servers. In *IEEE Symposium on Security and Privacy*, pages 100–114, Oakland, CA, US, May 2006. IEEE Computer Society.

[7] Steven J. Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, Oakland, CA, US, May 2005. IEEE Computer Society.

[8] D. Towsley S. B. Moon, P. Kelly. Estimation and removal of clock skew from network delay measurements. Technical Report 98-43, Department of Computer Science, University of Massachussetts at Amherst, October 1998.

[9] D. Mills. Network Time Protocol (Version 3) Specification, Implementation. RFC 1305, IETF, March 1992. `http://www.ietf.org/rfc/rfc1305.txt`.

[10] MaxMind. GeoLite Country, 2008. `http://www.maxmind.com/app/geoip_country`.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, IETF, May 1996. `http://www.ietf.org/rfc/rfc1945.txt`.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[13] Apache Software Foundation. Apache web server. `http://www.apache.org/`.

[14] C. Langton. Unlocking the phase lock loop - part 1, 2002. `http://www.complextoreal.com/chapters/pll.pdf`.

[15] Carnegie Mellon University. Pid tutorial. `http://www.engin.umich.edu/group/ctm/PID/PID.html`.

[16] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3), July 2003.

[17] S. Clowes. Tsocks - a transparent socks proxying library. `http://tsocks.sourceforge.net/`.

# NetAuth: Supporting User-Based Network Services

Manigandan Radhakrishnan
*mani@rites.uic.edu*
*University of Illinois at Chicago*

Jon A. Solworth
*solworth@rites.uic.edu*
*University of Illinois at Chicago*

## Abstract

In User-Based Network Services (UBNS), the process servicing requests from user $U$ runs under $U$'s ID. This enables (operating system) access controls to tailor service authorization to $U$. Like privilege separation, UBNS partitions applications into processes in such a way that each process' permission is minimized. However, because UBNS fundamentally affects the structure of an application, it is best performed early in the design process.

UBNS depends on other security mechanisms, most notably authentication and cryptographic protections. These seemingly straightforward needs add considerable complexity to application programming. To avoid this complexity, programmers regularly ignore security issues at the start of program construction. However, after the application is constructed, UBNS is difficult to apply since it would require significant structural changes to the application code.

This paper describes easy-to-use security mechanisms supporting UBNS, and thus significantly reducing the complexity of building UBNS applications. This simplification enables much earlier (and hence more effective) use of UBNS. It focuses the application developer's attention on the key security task in application development, partitioning applications so that least privilege can be effectively applied. It removes vulnerabilities due to poor application implementation or selection of security mechanisms. Finally, it enables significant control to be externally exerted on the application, increasing the ability of system administrators to control, understand, and secure such services.

## 1 Introduction

Computer networking was designed in a different era, in which computers were kept in locked rooms and communication occurred over leased lines, isolating systems from external attackers. Then, physical security went a long way in ensuring adequate computer security. Today, however, attackers can remotely target a computer system from anywhere in the world over the Internet.

Given that physical separation is no longer an alternative, securing networked applications requires isolation of a different form, including in general:

1. authentication of both users and hosts;
2. protection of communication confidentiality and integrity; and
3. authorization (also known as access controls) using least privilege [32].

The first two tasks are typically provided for within the application, for example by using SSL [12] or Kerberos [38]. The last task is ideally enforced by the Operating System (OS), since then failures in the application (e.g., a buffer overflow) do not bypass authorization.

But (1) and (2) are complicated by Application Program Interfaces (APIs) which are both difficult and tedious to use; for example, in addition to the basic authentication mechanism, it is necessary to communicate information from client to server (perhaps using GSSAPI [24]), interface to PAM [33], and the OS. The application programmer must choose from a large variety of authentication techniques (e.g., password or public-key), and compensate for their weaknesses. Since complexity is the enemy of security, it is especially important to avoid complexity in security critical code. And authentication is often attacked, for example, password dictionary attacks against SSH[1], as well as the implementations of authentication[2].

Consider the dovecot IMAP server. Over 9,000 lines are devoted to (1) and (2), consuming 37% of the IMAP service code (see Section 6 for details). Clearly, this is a large burden on application developers, and as we shall show, unnecessary. In contrast, the partitioning of the application into processes, and their attendant privileges, *is* a concern of application programmers since it is fundamental to program structure. The impact of this partitioning includes the number and purposes of processes, the privileges associated with processes, the communication between processes, the organization of data the processes access, the data and operations which must be performed within a process, the sequencing of operations, and the security vulnerabilities. For a general discussions of these issues, see [4].

One important way of partitioning network services is by the remote user $U$ they serve. That is, a server process which receives requests from $U$ runs under $U$'s user ID, so that its "ownership" is visible to, and limited by, external authorization. Although this scheme is widely used, we don't know a term for it, so we shall call it *User-Based Network Services (UBNS)*. UBNS is used, for example, in dovecot, SSH, and qmail. It prevents a user's private data from being commingled with other user's data and provides the basis for OS authorization. The latter enables system administrators to be able to configure secure services easily. Given the many sources of service code—and frequent releases of the services—it is highly desirable to move the security configuration and enforcement outside the service. This minimizes the harm that errant services can do, reduces the need to understand (often poorly documented) application security, enables strong protections independent of service code, is more resilient in the presence of security holes, and vastly increases the effectiveness of validating service security.

Despite the advantages of UBNS, authentication is often performed in a service-specific way or not at all. A prime example is the Apache web server (and most other web servers). In Apache, the users are not visible to the OS. The crucial independent check provided by OS-based authorization is lost. And application developers often avoid service-specific authentication, due to the complexity it engenders. Hence, an application's initial design often forgoes security concerns which then must be retrofitted after the fact [13]. But retrofitting UBNS requires restructuring and re-implementing substantial portions of the application. And since it is difficult to restructure existing applications, the service may never be made into a UBNS.

If UBNS were easier to implement at the application level, it could be integrated from the beginning of system design. Application complexity would be decreased and security would be improved. In this paper, we describe how to radically reduce complexity in UBNS service using *netAuth*—our network authentication and authorization framework. In netAuth, a service requires only 4 lines of code to implement authentication and 0 for encryption and authorization. Hence, netAuth

1. allows authenticated services to be easily integrated and
2. enables requests for the same user to be directed to the same back-end server.

The first is essential to support UBNS. The second makes it easier to re-use per user processes, removing the need for concurrent programming while increasing system efficiency. In addition, these mechanisms enable more modular construction of applications.

We describe NetAuth APIs and the implementations it gives rise to. By making these mechanisms almost entirely transparent, an application developer adds only minimal code to use these mechanisms. We describe sufficient networking interfaces to support UBNS and describe their implementations. These mechanisms are quite simple and thus are easy to use. The protections provided are also considerably stronger than those in most applications. We then describe a port of a UBNS service, dovecot to netAuth, and the substantial savings of code, simplifications to process structure, and reduced attack surface of this port.

The remainder of the paper is organized as follows: Section 2 describes related work. Section 3 describes the overview of our system. We then describe our system in more depth: Section 4 describes how our authentication mechanism can be used to write application. Section 5 describes briefly our implementation and some performance numbers. In Section 6, we describe the experience of porting dovecot to netAuth. Section 7 discusses the security achieved and finally we conclude.

## 2 Related work

UBNS is not the only way to partition a service into multiple processes. Another complementary way is *privilege separation* [29]—in which an application is partitioned into two processes, one privileged and one unprivileged. For example, the *listening* part of the service which performs generic processing—initialization, waiting for new connections, etc. is often run as `root` (i.e., with administrative privileges) because some actions need these privileges (for example, to read the file containing hashed passwords or to bind to a port). Unfortunately, exploiting a security hole in a root level process fully compromises the computer. By splitting the server into two processes, the exposure of a root level process is minimized. In contrast to UBNS, retrofitting privilege separation is not difficult, and there exists both libraries [20] and compiler techniques [6] to do it. Both UBNS and privilege separation are design strategies to maximize the value of least privilege [32].

SSH is a widely used UBNS service [42, 29], but is ill-suited to implement UBNS services—such as mail, calendaring, source control systems, remote file systems—because of the way network services are built. In the network case, the listening process exists before the connection is made and must at connect time know what user is associated with the service. SSH's port forwarding[3] performs user authentication at the service host—but not at the service—and hence, to the service the users of a host are undifferentiated[4]. As a result, traditional UBNS services use authentication mechanisms such as SSL or passwords and OS mechanisms such as setuid which are awkward to program and may not be secure. In contrast, netAuth both authenticates and authorizes the user on a per service basis, so that the service runs *only* with the permission of the user. Unlike SSH, netAuth provides end-to-end securing from client to service.

Distributed Firewalls [5] (based on Keynote [5]) in contrast to SSH, implements per user authorization for services by adding this semantics to the `connect` and `accept` APIs. While Distributed Firewalls sit in front of the service, and thus are not integrated with the service, Virtual Private Services are integrated and thus can provide UBNS services [16]. In DisCFS [27], an interesting scheme is used to extend the set of users on the fly by adding their public keys; although we have not yet implemented it, we intend to use this mechanism to allow anonymous access (assuming authorization allows it for a service) thus combining the best of authenticated and public services.

Shamon [25, 17] is a distributed access control system which runs on *Virtual Machines (VMs)*. It "knits" together the access control specifications for different systems, and ensures the integrity of the resulting system using TPM and attestation techniques. Its communication, like netAuth, is implemented in IPsec and uses a modified `xinetd` to perform the authorization. Shamon implements a very comprehensive mechanism for authorization (targeted for very tightly integrated systems), in contrast to netAuth's less complete but simpler service-by-service authorization.

We do not describe the authorization part of netAuth in this paper for two reasons. First, there is not sufficient space. Second, the authentication mechanism can be used with *any* authorization model. For example, even POSIX authorization, privilege separation, and VMs could be combined to provide a reasonable base for UBNS. The most value for authorization is gained when privileges are based both on the executable and the user of the process, increasing the value of privilege separation. Such separation is essential to allow multiple privilege separated services to run on the same OS. Examples of such mechanisms include SELinux [34], AppArmor [9], and KernelSec [30]. Janus[15], MAPBox[2], Ostia[14] and systrace[28, 22] are examples of sandboxing mechanisms which attenuate privileges.

SANE/Ethane [8, 7] has a novel method of authorizing traffic in the network. An authorizing controller intercepts traffic and—based on user authentication for that host—determines whether to allow or deny the network flow. This enables errant hosts or routers to be isolated. However, the authentication information available to Ethane using traditional OS mechanisms is coarse grain (it cannot distinguish individual users or applications). Ethane and netAuth are complementary approaches, which could be combined to provide network-based authentication with fine-grained authentication.

Distributed authentication consist of two components: a mechanism to authenticate the remote user and a means to change the ownership of a process. Traditionally, UNIX performs user authentication in a (user space) process and then sets the User ID by calling `setuid`. The process doing `setuid` needs to run as the superuser (administrative mode in Windows) [39]. To reduce the dangers of exploits using such highly privileged processes, Compartmented Mode Workstations divided root privileges into about 30 separate capabilities [3], including a SETUID capability. These capabilities were also adopted by the POSIX 1e draft standard [1], which was widely implemented, including in Linux.

To limit the setuid privileges further, Plan9 uses an even finer grain one-time-use capability [10], which allows a process owned by $U_1$ to change its owner to $U_2$. NetAuth takes a further step in narrowing this privilege since it is limited to a particular connection and is non-transferable; but a more important effect is that it is statically declared and thus enhances information assurance whether manually or automatically performed.

The traditional mechanism to provide user authentication in distributed systems is passwords. Such passwords are subject to dictionary and other types of attacks, and are regularly compromised. Even mechanisms like SSL typically use password based authentication for users [12] even though they can support public key encryption.

Kerberos [38] performs encryption using private key cryptography. Kerberos has a single point of failure if the KDC is compromised; private key also means that there is no non-repudiation to prove that the user did authenticate against a server; and requires that the KDC be trusted by both parties. Microsoft Window's primary authentication mechanism is Kerberos.

Plan9 uses a separate (privilege-separated) process called `factotum`, to hold authentication information and verify authentication. The factotum process associated with the server is required to create the change-of-owner capability. But factotum is invoked by the service, and hence can be bypassed allowing unauthenticated users to access the service. Of course, it is in principle possible to examine the source code for the service to determine whether authentication is bypassed, but this is an error prone process and must be done anew each time an application is modified. NetAuth, enforces authentication and authorization which cannot be bypassed and is easier to analyze.

The OKWS web server [21], built on top of the Asbestos OS [11] does a per user demultiplex, so that each web server process is owned by a single user. This in turn is based on HTTP-based connections, in which there can be multiple connections per user, tied together via cookies. It uses the web-specific mechanism for sharing authentication across multiple connections. OKWS was an inspiration for netAuth, which allows multiple connections from a user to go to the same server. NetAuth works by unambiguously naming the connection so that it works with any TCP/IP connection; and hence is much broader than web-based techniques.

## 3 System overview

NetAuth is modular, so that the different implementations and algorithms can be used for each of the following three components:

1. **User authentication** is triggered by new network APIs which (a) transparently perform cryptographic (public key) authentication over the network and (b) provide OS-based ownership of processes. Part (b) inherently requires an authorization mechanism which controls the conditions under which the user of a process can be changed.

2. **Encrypted communication between authenticated hosts** ensures that confidentiality and integrity of communications are maintained, and also performs host authentication. This encryption is provided by the system and requires *no* application code.

3. **Authorization** is used to determine if a process can (a) change ownership, (b) authenticate as a client, (c) perform network operations to a given address, and (d) access files (and other OS objects). In UBNS this ideally depends on both the service and the user. Thus, the authentication mechanism essentially labels server processes with the user on whose behalf the service is being performed so that external authorization can be done effectively. It is highly desirable that the authorization system prevent attacks on one service spilling over to other services.

Due to space limitations, this paper focuses on user authentication. Authentication may seem trivial, but it requires significant amount of code in applications, so much so, that this mechanism is justified solely to improve authentication (without also improving authorization). Our server implementation is in the Linux kernel, but our client is user-space code which can be ported to any OS, including proprietary ones.

For encryption we require that hosts be authenticated and that cryptographic protections be set up transparently between hosts. Host authentication is important since if the end computer is owned by an attacker then security is lost. Such end devices can be highly portable devices such as cellphones. (For less important application one can use untrusted hosts.) Encryption can be triggered either in the network stack or by a standalone process. Currently, we are using IPsec [19, 18] for this purpose as recent standards for IPsec have made it significantly more attractive as it allows for one of the hosts to be NATed [40]. But we expect to replace it with a new suite being developed which will be far less complex and faster.

The netAuth API can be used with any authorization model, which would need to control both change of ownership and client authentication, perhaps using simple configuration files [20] as well as networking and file systems to some extent. NetAuth's authorization model controls who may `bind`, `accept`, and `connect` to remote services on a per user basis as well as fine-grain support for the user and services which can access a file. NetAuth's authorization model is fully implemented, and we will describe it in a forthcoming paper.

A central tenet of our design is a clear separation between administration and use of our system. Even when the same person is performing both roles, this separation enables allowed actions to be determined in advance, instead of being interrupted in mid-task with authorization questions (e.g., "do you accept this certificate?"). It also supports a model of dedicated system administrators; further partitioning of the system administration task is possible, for example to allow outsourcing of parts of the policy.

In netAuth, user processes never have access to cryptographic keys and cryptographic keys can only be used in authorized ways. Hence, from the authorization configuration the system administrator can easily determine which users are allowed to use a service and how services can interact with each other.

NetAuth enables successive connections by the same user to be directed to a single process dedicated to that user. We shall see that this has both programming and efficiency advantages. In addition to its uses in traditional network services, it can be used to easily set up back ends on the same system, and thus allow for further opportunities for UBNS.

We next give an overview of network authentication and UBNS mechanisms in netAuth.

**Network authentication** netAuth enables the owner of a process to be changed upon successful network authentication. Authentication is implemented as follows:

- the server system administrator must enable UBNS change-of-ownership by specifying the *netAuthenticate* privilege for the service.
- the client process requests the OS to create a connection and a time-limited connection-specific digitally signed authenticator[5] [31].
- the server process explicitly requests the OS to perform network authentication. The user authentication is only usable by the designated server process (it is non-transferable).

This mechanism requires that the client-side system administrator enable the client to use netAuth authentication, and the server-side administrator provide the netAuthenticate privilege. As we shall see, application code changes to support authentication are trivial on both client and server sides.

Because public key signatures are used for authentication, the log containing these signed exchanges proves that the client requested user authentication. This property both helps to debug the mechanism and to ensure that even the server administrator cannot fake a user authentication. Lastly, since no passwords are used over the network, this scheme is impervious to password guessing attacks.

**UBNS** netAuth has a built-in mechanism to support UBNS. All connections to a specified service from user $U_i$ can be served by a single server process $p_i$ unique to that user. For users $U_j$, for which there does not exist a corresponding process $p_j$, a listening process $p$ pre-accepts (see Section 4) the connection and creates a new process $p_j$[6]. Figure 1(a) shows two types of queues of unaccepted connections maintained by netAuth (one for new users and the other for users for which there exists a user process).

Per user server processes are created on demand for efficiency and flexibility. Successive connections for $U_i$ will reuse server process $p_i$. NetAuth can also support other commonly used methods such as pre-forking processes or forking a process per connection.

This mechanism provides a very clean programming model as it is trivial to create back-end services for each user on demand. For example, Figure 1(b) shows a calendar proxy which caches a user's local and remote calendars (and no one else's) and provide feeds to a desk planner, email to calendar appointment program, a reminder system, etc. The reminder mechanism might know where the user is currently located and where the appointment is, so that reminders can be given with suitable lead times. As the user's connections are always to the same process, requests are serialized for that user preventing race condition (and the need to synchronize) and enable easy adding of calendar applications without configuring for security (since the configuration is in the proxy). Such a model also allows different parts of the application to execute on different systems. For example, a user interface component could run on a notebook, and a backend store could run on an always available server.

We next look at the uses of NetAuth in more detail.

## 4 NetAuth Application Programming Interface

There are several ways to set the owner of a network service: (1) the service can be configured to run as a pseudo user (e.g., apache) with enough privileges to satisfy any request. (2) the service may need user authentication to ensure that it is a valid user (e.g., for mail relay), but all users are treated identically. This service too can be owned by a pseudo user. (3) the service provided depends on the user, who therefore must be authenticated—it is usually appropriate that the service process be owned by its user (i.e., UBNS).

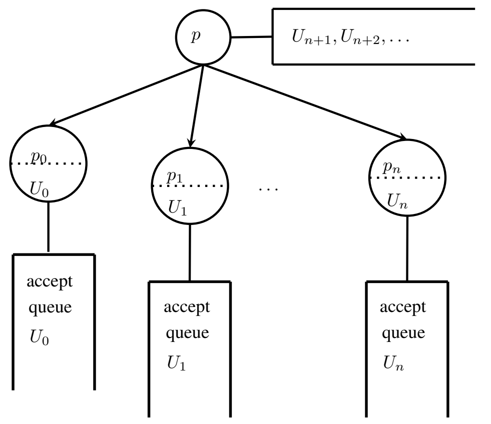
A UBNS service (a process run under the user's ID) performs the following steps: (a) it accepts a connection, (b) performs user authentication to identify the user requesting the service, (c) creates a new process, and (d) changes the ownership of the process to the authenticated user. Once the ownership of the process is changed to the user, it cannot be used by anyone else.

We next examine how this general paradigm is performed in Unix and then in netAuth.

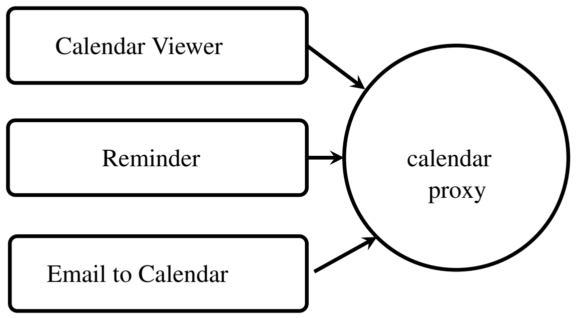
Figure 2(a) shows the call sequence for implementing a user authenticated service using UNIX socket APIs. The client creates a socket (`socket`), connects to the server (`connect`), and then does a series of sends and receives (`send/recv`), and when its done closes the socket.

The server creates a socket (`socket`), associates it with a network address on the server (`bind`), allocates a pending queue of connection requests (`listen`), waits for a new connection request to arrive (`accept`). To perform UBNS, it spawns a process (`fork`), and after determining the user via network messages (not shown) it then changes the owner of the process (`setuid`). At this point the newly created service process is operating as the user. It communicates back and forth with the client and then closes the connection. Since there is typically no way to reuse the process after it closes the socket, it exits.

Figure 2(b) shows the equivalent sequencing for netAuth. On the client side, the only programming change needed to adapt to netAuth is to replace `connect` with `connect_by_user` (of course, the application-level authentication must be removed).

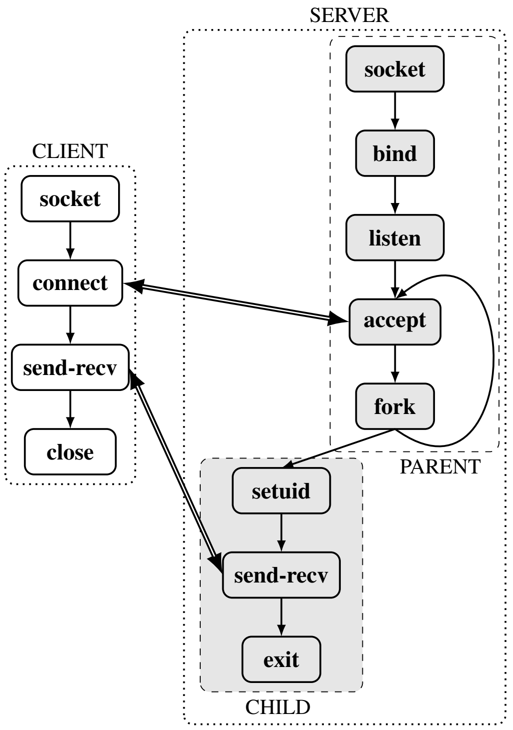


(a) mapping all the connections of a user to the same process

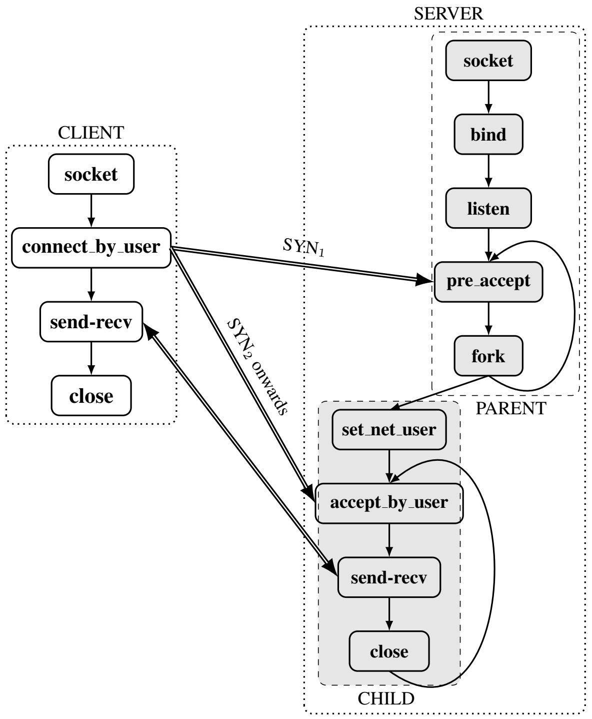


(b) calendar privilege separation example

Figure 1: Privilege separation in netAuth



(a) UNIX



(b) NetAuth

Figure 2: Sequence of system calls executed by a client and a server. The server forks a process to service a request; the forked process is owned by the authenticated user.

On the server side, netAuth basically splits the accept for a new connection into two phases:

- The first phase is called the `pre_accept`, which determines when a new user (one that does not have a service process) arrives. Hence, the `pre_accept` blocks until there is a waiting connection for some user $U$ without a corresponding service process owned by $U$. (To prevent race conditions, a process which has a temporary reservation for $U$ by virtue of having done a `pre_accept` but not yet having changed the owner is reserved by $U$.)

- The second phase is the `accept_by_user` to actually accept the connection, after having created a process owned by the new user.

The accept is split into two APIs because there are now two actions (1) determining that there is an unaccepted connection for a new user (so that a new process can be created) and (2) completing the accept by a (child) process owned by the new user. (2) ensures that the accepted socket can be read or written (since the process is owned by the user). Hence, the split accept ensures that the `accept_by_user` only succeeds if the owner of the process is the authenticated user on the connection.

The change of ownership of the process is performed by `set_net_user`. The `set_net_user` changes the owner of the process to the authenticated user and consumes the `netAuthenticate` privilege for that process. Thus, `set_net_user` serves as a highly restricted version of `setuid`, and is far safer to use.

## 5 Implementation

In this section, we describe the netAuth architecture, the protocol for user authentication, and the implementation. We then describe some performance numbers.

### 5.1 Architecture

The design of netAuth emphasizes the separation of authentication, authorization, and cryptographic mechanisms away from the application.

The overall architecture is shown in Figure 3. Applications communicate with each other using APIs which emphasize process authentication—the one component of netAuth which must be visible to networked application code. There are two types of communications, both of which flow over an IPsec tunnel between the hosts:

- the application's protocol (or data, for performing its function) and

- the netAuth authentication information.

The authentication information is managed by two netAuth daemons—netAuthClient and netAuthServer—which perform both the public key operations for user authentication and enable the process' change of ownership.

### 5.2 Authentication protocol

Because IPsec is used for communication, IPsec performs host authentication. This means that the remote service is authenticated, because the service type is determined by port and the IP is verified using IPsec's public key host authentication.

Before application communication is established, user authentication is performed:

**netAuthClient** signs an authenticator which describes the connection.

**netAuthServer** receives the authenticator and verifies its signature.

Public-key cryptographic operations can be considerably more expensive than symmetric key algorithms. Fortunately, signing (which is done on the relatively idle client) takes significantly longer than verifying (on a busy server). For example, RSA public key signing times (client) and verification times (server) for 1024 and 2048 bit keys are shown in Table 1[7].

Once the `netAuthClient` has proved that it can sign the authenticator, successive signings prove little (since from the first signing we know that the netAuthClient has the requisite private key). Hence, successive connects for that user employ a quick authentication based on hash chains [23].

We use a separate connection to send our authenticator, rather than the more traditional mechanism of piggybacking authentication on the application connection. This is done both to increase the flexibility of communications and to allow connections to be re-authenticated periodically. Re-authentication determines whether the user's account is still active, and hence a re-authentication failure disables the user's account and stops their processes, something that is difficult to do with other protocols. We re-authenticate using the same hash chain scheme as for successive connects for the same user.

| **key size** | **signing** | **verifying** |
|---|---:|---:|
| 1024 | 680 $\mu$s | 40 $\mu$s |
| 2048 | 2,780 $\mu$s | 80 $\mu$s |

Table 1: RSA signing/verification times in $\mu$seconds

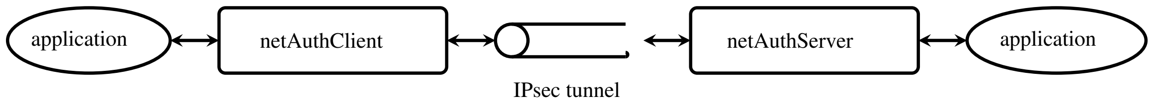


Figure 3: Architecture of netAuth

## 5.3 kernel-based implementation

The first NetAuth implementation has been integrated into the Linux kernel. Our implementation has three key components:

- **kernel extensions** (code integrated into the mainline kernel) implementing networking support for processes with per-user privileges and providing the new system calls `pre_accept`, `set_net_user` and `accept_by_user`.

- a **loadable kernel module** implementing netAuth authorizations, uses the Linux Security Module (LSM) framework [41].

  The LSM framework segregates the placement of hooks (scattered through the Linux kernel) from the enforcement of access controls (centralized in an LSM module). Thus changes in the mainline kernel (mostly) do not affect LSM modules.

- Three **user-space daemons** which (1) download the networking policy into the kernel using the `netlink` facility (2) sign authenticators and (3) verify authenticators.

The kernel implementation currently consists of about 3,700 lines of C code (∼3,000 in the kernel module and ∼700 in the kernel extensions).

## 5.4 Performance

We now report on NetAuth's performance. All the experiments were run using a server—an AMD 4200+ (2.2 GHz) machine with 2GB RAM—and a client—an AMD 4600+ (2.2 GHz) machine with 1GB RAM. Both computers ran Linux kernel v2.6.17, used gigabit networking, and were connected by a crossover cable[8]. We measured elapse times (from the applications) in all cases.

We performed two types of performance tests to measure latency. First, we measured the overhead of netAuth authorization and compared it to unmodified Linux, for the cases of the `bind`, `connect` and `connect-send-recv` operations. Second, we measured latency for netAuth's per-user services. For the second part, there is no comparable Linux scenario and hence we report absolute times there.

| | UNIX ($\mu$s) | NetAuth ($\mu$s) | Overhead |
|---|---|---|---|
| `bind` | 6.00 | 6.75 | 12.5% |
| `connect` | 28.00 | 32.00 | 14.28% |
| `connect-send-recv` (Unix style) | 145.00 | 157.00 | 8.27% |

Table 2: Elapse times for the micro-benchmarks and the Unix-style concurrent server (see Section 3). No authentication is performed in any of these cases. The time specified are all in micro seconds.

### 5.4.1 System call overhead (no authentication)

Our first measurements determine the authorization overheads for netAuth, by using a light weight authentication with minimal overhead. The authorization mechanism limits which users can use the service, it is implemented outside of application.

The measurements are given in Table 2, are of netAuth vs. unmodified Linux:

- the time to perform a `bind` by a server increased by 12.5% due to the overhead of doing the authorization checks.

- the time to complete a `connect` (as measured) on the client-side increased by 14.28%, due to client-side and server-side authorization checks. The elapsed time includes a round trip packet time.

- the time to do a `connect-send-recv` (as measured) on the client is considered next. (The server must do a `accept-fork-setuid-recv-send`). The send and recv are 128 bytes of data. For the UNIX case, the total time was about 145 $\mu$ seconds while for the NetAuth case the time was about 157 $\mu$ seconds, an overhead of 8.27%. The most costly operation is the `fork` performed at the server to create a new per-user process.

We note that these overheads are best case [26], normally latency issues are higher. Moreover, no performance tuning has yet been done on the netAuth implementation.

### 5.4.2 Using netAuth authentication

This section describes the case of a server in which the process for user $U_i$ satisfies *all* of the requests from $U_i$, for a particular service (as described in Section 5). There does not exist a comparable scenario in UNIX. Hence, we report the latencies observed on the client side in Table 3.

| Connection | netAuth (with auth.) | Linux (w/o auth.) |
|---|---|---|
| first | 4200 $\mu$s | 147 $\mu$s |
| successive | 67 $\mu$s | 147 $\mu$s |

Table 3: Elapse times observed on the client side to perform a `connect-send-recv`. The netAuth connections are established with user authentication. Successive netAuth connections are to the same per-user server process created by the first connection on the server. The UNIX connections are established without user authentication.

For the first connection, using netAuth authentication mechanisms, a new connection results in the following set of actions: (1) on the client, the kernel requests an authenticator from the user-space daemon; (2) the client generates the authenticator and sends it to the server where it is verified; (3) there is a RTT for sending the authenticator to the server and receiving response from the server; (4) there may be context-switch times (between client process and authentication daemon); and (5) there may scheduling delays. The costliest operation by far is the cryptographic signing of the authenticator.

All subsequent connections on behalf of the same user run much faster because they re-use the same server process and fast authentications. In comparison, the elapse time for the UNIX case is the same for all cases because there are no schemes for a client to re-use a previously created per-user process. The values for UNIX shown in the Table are without authentication overhead.

#### 5.4.3 Server throughput

We next consider server throughput in terms of new connections. In netAuth, although the first authentication must be signed, successive authentications require only a very fast cryptographic hash. From table 1, the service-based verification of signatures takes only 80$\mu$seconds. Hence, a single core can perform authentication for 45,000,000 users per hour assuming authentications are cached for one hour. We believe that such performance levels eliminate the need to consider weaker authentication mechanisms, even for very high volume services.

### 5.5 Alternative implementations

Our first implementation, which is described here, is a kernel-level implementation. Of course, we would like the APIs described here to be available on other systems without kernel modifications, particularly for those OSes for which the source code is not available.

We consider here only the client side issues as we would like netAuth services to be usable from any operating system. (Server OS, on the other hand, is under the control of the service provider.) In section 6.2 we describe a proxy implementation which uses netAuth, but which could be easily extended into one which implemented netAuth at the protocol level from user space rather than using netAuth APIs.

## 6 Porting applications to netAuth

To show the effectiveness of netAuth we ported a UBNS service. We have not yet attempted to port a service which is not UBNS organized (such as Apache), as that is a far more difficult problem. We chose an application, dovecot, which supports both privilege separation and UBNS.

| Process name | executable name | user ID |
|---|---|---|
| `master` | dovecot | root |
| `auth` | dovecot-auth | root |
| `login` | imap-login<br>pop3-login | dovecot<br>dovecot |
| `imap` | imap | $U$ |

Table 4: Dovecot processes and their respective user ID's. Here $U$ refers to the user ID of the (remote) user whose is accessing her mail.

Dovecot is an open source IMAP and POP mail server (and is included in Linux distributions such as Debian and Ubuntu). Users can access dovecot-based services remotely using a *Mail Viewer Agent* (`MVA`) such as Thunderbird or Outlook. The `MVA` on the client communicates with dovecot using the IMAP or POP protocols over SSL or unencrypted connections.

Dovecot was built with security as a primary goal. Since January 2006, its developer has offered an as-yet-uncollected reward of 1000€ for the first provable security hole[9]. To support both privilege separation and UBNS, dovecot has four process types, running under root, dovecot pseudo user, and the user $U$ retrieving her mail, as shown in Table 4.

Table 5, shows the code organization of the dovecot distribution supporting IMAP (v1.0.9)[10]. Dovecot also uses pam, crypto, and ssl libraries which are not included in these line counts. The source distribution to support IMAP is 24,628 lines of code, of which 9,307[11] (37.8%) are associated with authentication and encryption. The port consisted of removing this code, and copying over less than 1,000 lines from `master` (configuration and the concurrent server loop) and `login` (the initial handshake code) to `imap`.

The port reduces the number of process types from four to one. With a traditional Unix authorization model, the port still requires root to bind to port 143 and to do setuid; but unlike the pre-port version, our `imap` process never reads user input while running as root and thus is not subject as root to buffer overflow attack. (The privileges can be still reduced further using netAuth's authorization model).

When implementing a `imap` service from scratch, only 4 netAuth specific lines would be needed to provide authentication and encryption over that required for an unauthenticated service.

## 6.1 Dovecot before and after

The standard version of dovecot is more complex because of the privilege separation mechanism and especially the complexity of using standard authentication and cryptographic mechanisms. We describe first the processes and then later the operations needed to retrieve IMAP mail in standard dovecot.

### 6.1.1 Standard dovecot

The dovecot distribution is composed of the following processes:

**`master` process** starts the `auth` process and $n$ (by default, 3) `login` processes. The `master` process is also responsible for the creation of an `imap` process after a successful authentication.

**`auth` process** authenticates new users for the `login` process (over a UNIX socket). The `auth` process also verifies successful authentications to the `master` before it creates a `mail` process.

**`login` process** listens on the appropriate port (e.g., 143 for IMAP) for new connections. Once a connection is established it negotiates with the `MVA` process to initialize the connection (sending server capabilities, setting up SSL, etc.) and requests authentication of the user. Upon successful authentication, the `login` process requests the `master` process to create a new `imap` process and then exits.

**`imap` process** receives the socket descriptor over a UNIX socket from the `login` process. The `imap` process then communicates with the remote `MVA` to access the user's mailbox on the server.

Figure 4 shows the sequence of events that are necessary to create a new `imap` process to service requests from the `MVA`.

1. Messages 1a and 1b establish the initial connection between the `MVA` and dovecot. During this step, the `MVA` requests and receives the server capabilities (not shown in the figure).

2. authentication step (shown as messages 2a-2e and action 2f). (a) The `MVA` sends the user's authentication information as part of a LOGIN message. (b) In response, the `login` process requests the `auth` process to authenticate the user (c) The `login` process request the `auth` process to authenticate, (d) on successful authentication the `login` process sends a response back to the `MVA` and (e) requests the `master` process to create a new `imap` process. (f) the `master`, after verifying a successful authentication with the `auth` process, creates the new `mail` process running on behalf of user $U_k$.

3. The `imap` process then services the `MVA`'s future requests.

### 6.1.2 Porting dovecot to netAuth

The porting of dovecot to netAuth consists of (a) removing code, (b) moving some code into the `imap` process and (c) removing three of the four processes. A dovecot process ported to netAuth is not expected to perform the following functions: message encryption using OpenSSL, GNU-TLS or the like; user authentication; performing the complex `setuid()` operation and related code to ensure that the process does not have any privileged left-overs (in the form of file descriptors) in the unprivileged process. Hence, code for these security sensitive operations need not be implemented by the dovecot executable and can be removed. Thus, summarizing the dovecot port to netAuth:

- the `auth` process (and its code) is eliminated completely as the user authentication is performed by the OS as part of connection establishment.

- from the `master` process, only the code to `bind` to the privileged port and to configure a new mail process with the appropriate set of environment variables is retained. (Dovecot passes configuration information to the `imap` process as environment variables)

- from the `login` process, only the initialization of a new connection (1a and 1b in Figure 4) is retained.

- the core functionality of accessing and maintaining mailboxes in the `imap` process is retained.

Thus, the dovecot port to netAuth runs as a single process type (following the design for a concurrent server implementation shown in Figure 2). The `master`, `auth`

| directory | total lines of code |
|---|---:|
| master | 2,460 |
| auth | 5,469 |
| imap-login | 484 |
| imap | 3,456 |
| lib-auth | 490 |
| lib | 6,268 |
| login-common | 1,138 |
| lib-imap | 1,069 |
| lib-settings | 101 |
| lib-ntlm | 304 |
| lib-sql | 882 |
| lib-dict | 470 |
| lib-storage | 574 |
| lib-mail | 1,463 |
| total | 24,628 |

| process | dovecot's libraries used | total lines of code | dynamic libraries |
|---|---|---|---|
| master | lib | 8,728 | |
| auth | lib, lib-settings, lib-ntlm, lib-sql | 13,024 | pam crypto |
| login | lib, login-common, lib-auth, lib-imap | 9,449 | ssl crypt |
| imap | lib, lib-dict, lib-mail, lib-imap, lib-storage | 13,300 | ssl |

Table 5: Table with lines of code in the various directories in dovecot. The command '`cat *.c *.h | grep ";" | wc -l`' was used to determine this count.

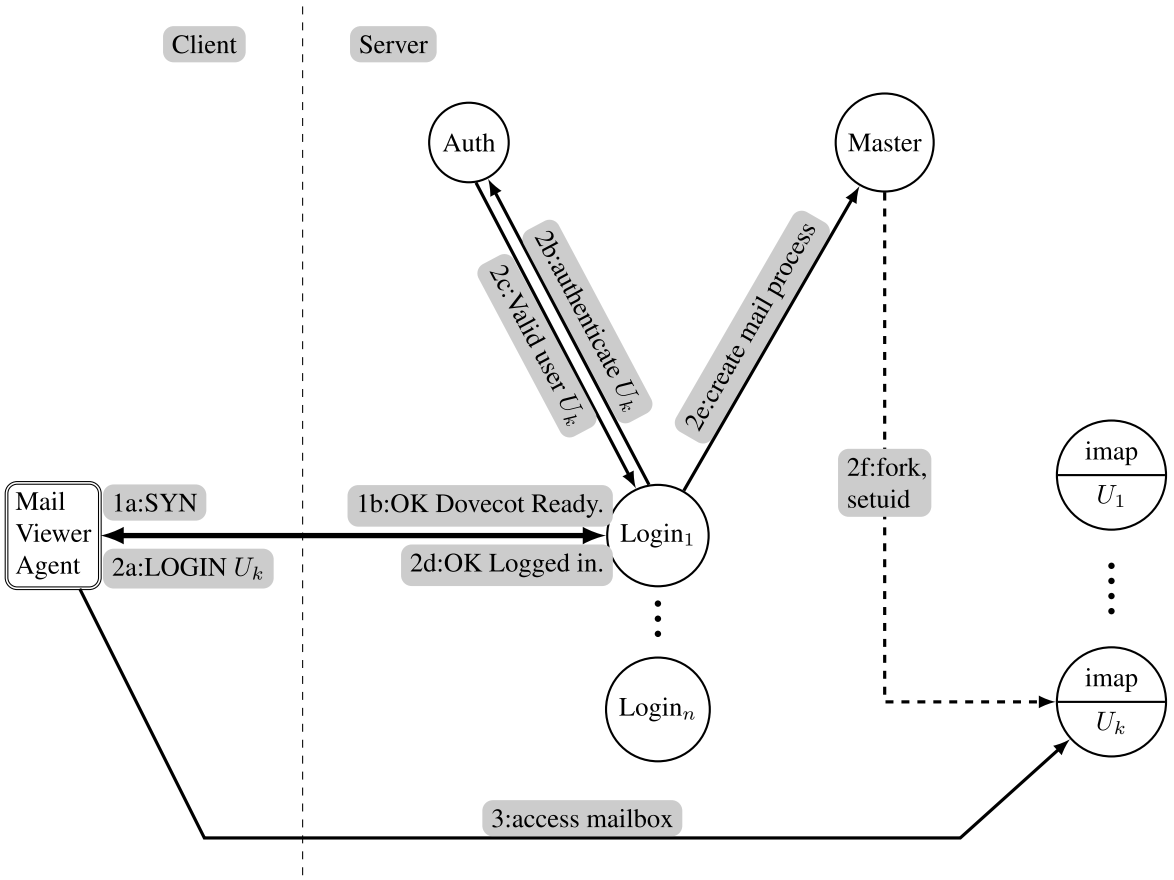


Figure 4: The processes that comprise standard dovecot and their interaction to authenticate a user. Solid arrows indicate message exchange while dashed ones represent process actions. Message exchange across system boundaries use a network socket while those within the same system use UNIX sockets.

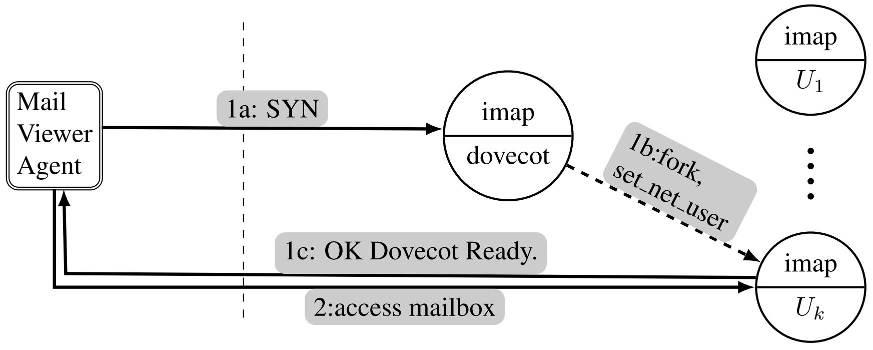


Figure 5: The message exchanges between the ported netAuth dovecot and the MVA.

and `login` processes are eliminated after taking a small amount code from them.

The resulting `imap` code performs the following steps:

- initializes a socket to listen for new connections. It performs a `bind` on the privileged port, a `listen`, sets the accept mode to acceptByUser and blocks on `pre_accept` waiting for a connection from a user for which there is no `imap` process.
- when a connection from a new user arrives, the process returns from `pre_accept` with the new user's information. The process forks a child process to handle the user and returns to waiting for a new user.
- the child process changes the user by executing `set_net_user` with the user information from the `pre_accept` call. The child process runs as the new user. This process can now accept the connections (for that user) and process the MVA's requests.

The user is authenticated as part of the processing in the network stack to accept a connection Hence, `pre_accept` returns only for authenticated users. Connection requests of users that fail to successfully authentication are dropped (with a RST sent back).

## 6.2 Client side modifications

To test out the server-side modifications it was necessary to produce a netAuth-enabled MVA. Rather than port an existing MVA, such as Thunderbird, we instead built a netAuth proxy. This has several advantages, including portability to systems which do not allow kernel modifications and ability to support a wide variety of MVAs without doing multiple ports. The proxy presented the least invasive approach.

The proxy binds to the IMAP (or POP) port on the `localhost`. The events to setup a new connection:

- proxy binds to the privileged IMAP port and waits blocking for connection request.
- when a new connection request comes in from the MVA, the proxy authenticates the MVA. Once authenticated, the proxy initiates communication with the dovecot server using the `connect_by_user` system call.
- Once connected, the proxy just forwards messages to and from the MVA.

**Multiple dovecot servers** It is not unusual for a user to have multiple mailboxes maintained at more than one server. In this case, the proxy maintains a system-wide mapping (common to all users) from non-routable local IP addresses in the range 127.0.0.0/8 to the well-known routable IP address of the remote host running the dovecot server. All the MVA's on the client are then configured to use IP addresses in this range (published by the proxy) to refer their respective hosts.

The proxy binds and listens for connection requests on all the published local interfaces (i.e., all the 127.0.0.0/8 IP addresses configured for the proxy). A request on a given IP address corresponds to a particular remote host (known to the proxy). The proxy can then follow the scheme outlined above to authenticate the user and establish the connection.

## 7 Security achieved

The user never has access to his private keys, and in fact needs permission to authenticate using the private keys. This mechanism can be expanded to allow different private keys for different uses, although we do not yet support that. One use of such a facility is to allow the user to perform personal chores, such as banking with one key and to perform business functions with another key.

Only the specified users can connect to the service, since they must be authorized. This authorization is independent of a service; if the service is designated as an authenticated and authorized service, there is nothing the service can do (either deliberately or accidentally) to evade this mechanisms. The process may avoid actually setting the user ID, but the mechanism pairs user authentication with the connection, so that it can only be used by the process which accepts that connection *and* it is necessary to authenticate before reading or writing to the connection.

Because the authorization and authentication of user services are totally declarative, it is possible to automatically analyze them. (In contrast, this is not possible in general, due to decidability problems, when these functions are performed by application code.) We are planning on extending our previous work in DAC and MAC access controls to automatically analyze authorization properties *across* computer systems [36, 35, 37].

## 8 Conclusion

UBNS requires a mechanism for (1) authentication of users over the network and (2) allowing server processes to change the user on whose behalf they execute. Implementing the cryptographic mechanisms for user authentication as part of the application is complex and error prone, and as we showed, requires a substantial amount of code. Moving the authentication and cryptographic mechanisms outside the application makes the application independent of these mechanism, and application programmers are usually not skilled in this area. Moreover, the OS mechanisms for change of process ownership are also dangerous as such privileges are among the strongest in a computer system, since changing a user typically allows the privileges of *any* user to be appropriated.

And hence, programmers typically defer such considerations, ignoring them during initial design. But UBNS affects the very structure of programs and when its consideration is delayed, it becomes increasingly expensive to retrofit. Thus many applications will not be structured as UBNS and the design will not satisfy the property of least privilege.

NetAuth is a simple mechanism to invoke network authentication and process change-of-ownership, thus encouraging the design of UBNS. It builds on the work of Kerberos, SSH, and Plan9 but seeks to do so with the style of mandatory access controls and to provide better information assurance. It

- Requires only four lines of code for authenticated and cryptographically protected communications vs. a (concurrent) service which neither authenticates nor encrypts traffic.
- Enables the application developer to focus on the key task of partitioning the application into processes early in the design process.
- Remove the need for privileged processes to receive external input, and thus guards against a range of attacks including buffer overflow.
- Makes application code independent of the authentication method, thus enabling changes in the authentication methods without affecting either source or binary code.
- Externalizes authorization, making it independent of application failures.

While the authentication mechanism and APIs described here can be used with *any* authorization model, we have also built an authorization model (to be described elsewhere) which has a highly analyzable configuration in which strong properties can be understood independently of the application code.

NetAuth integrates public key and a fast reauthentication mechanism to achieve high performance authentications with the strongest possible properties. Further increases in performance are enabled by the reuse of processes for the same user, saving system overhead. This simplifies the structure of such applications, and makes it much easier to build UBNS. Such an easy-to-use mechanism will encourage programmers to integrate security from the start, and thus construct more secure applications.

Not only do these mechanisms enable the construction of more secure services but also provide significant advantages for system administration. These mechanisms enable strong controls to be imposed on services without resorting to application specific configuration and without analyzing application code.

## Acknowledgements

The authors would like to thank Jorge Hernandez-Herrero and the anonymous referees for their valuable feedback.

This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

## Notes

[1]`http://www.securityfocus.com/infocus/1876`
[2]`http://www.dovecot.org/security.html`

[3]Alternatively, SSH allows a remote executable to be invoked, but that remote executable is not connected to as a network service.

[4]hg-login `http://www.selenic.com/mercurial/wiki/index.cgi/SharedSSH`, as used in Mercurial, performs remote authentication using SSH, but execs a new program rather than connect to a running network service.

[5]We are using a simplified, and easily customized, certificate rather than the complex X.509 certificates.

[6]The application code forks the new process $p_j$. This explicit structure allows also non-privilege-separated iterative and concurrent service, although these exist largely for legacy applications.

[7]Source `http://www.cryptopp.com/benchmarks-amd64.html`, for an AMD Opeteron 2.4 GHz processor

[8]The server has an nVidia 570 chipset and the client an nVidia 430 chipset. They both run the open source `forcedeth` driver.

[9]The webpage at http://www.dovecot.org/security.html displays a list of security holes found in dovecot since the announcement of the award. The dovecot developer (maintainer of the webpage) claims that these holes cannot be exploited under reasonable circumstance stated as a set of rules on the same page.

[10]Dovecot also supports POP, which we ignore for this comparison.

[11]Code from the directories: auth, imap-login, login-common, lib-auth and master (except the configuration code).

## References

[1] IEEE/ANSI Draft Std. 1003.1e. Draft Standard for Information Technology–POSIX Part 1: System API: Protection, Audit and Control Interface, 1997.

[2] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, August 2000. USENIX.

[3] Jeffrey L. Berger, Jeffrey Picciotto, John P. L. Woodward, and Paul T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, 1990. Special Section on Security and Privacy.

[4] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *First Computer Security Architecture Workshop*, page 1. ACM, 2007. Invited paper.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. RFC 2704: The KeyNote Trust-Management System Version 2, September 1999.

[6] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In Jun Murai and Kenjiro Cho, editors, *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12. ACM, August 2007.

[8] Martin Casado, Tal Garfinkel, Aditya Akella, Michael Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. Sane: A protection architecture for enterprise networks. In *Usenix Security*, October 2006.

[9] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain: Parsimonious security server. In *14th Systems Administration Conference (LISA 2000)*, pages 355–367, New Orleans, LA, 2000.

[10] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In *Proc. of the USENIX Security Symposium*, pages 3–16, 2002.

[11] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005.

[12] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol — version 3.0. Internet Draft, Transport Layer Security Working Group, November 1996.

[13] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symposium on Security and Privacy*, pages 214–229, 2006.

[14] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, February 2004.

[15] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. of the USENIX Security Symposium*, San Jose, Ca., 1996.

[16] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Virtual private services: Coordinated policy enforcement for distributed applications. *IJNS*, 4(1), January 2007. `http://www1.cs.columbia.edu/~angelos/Papers/2006/ijns.pdf`.

[17] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, August 2006.

[18] C. Kaufman. RFC 4306: Internet key exchange (ikev2) protocol, December 2005.

[19] S. Kent and K. Seo. RFC 4301: Security architecture for the internet protocol, December 2005.

[20] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284. USENIX, 2003.

[21] Maxwell N. Krohn. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track*, pages 185–198, 2004.

[22] Aleksey Kurchuk and Angelos D. Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *SEC*, pages 473–488, 2004.

[23] Leslie Lamport. Password authentification with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[24] John Linn. Generic interface to security services. *Computer Communications*, 17(7):476–482, July 1994.

[25] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramón Cáceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *ACSAC*, pages 23–32. IEEE Computer Society, 2006.

[26] Larry McVoy and Carl Staelin. `lmbench`: Portable tools for performance analysis. In USENIX, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 279–294, pub-USENIX:adr, 1996. USENIX.

[27] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Secure and flexible global file sharing. In *USENIX Annual Technical Conference, FREENIX Track*, pages 165–178. USENIX, 2003.

[28] Niels Provos. Improving host security with system call policies. Technical report, CITI, University of Michigan, 2002.

[29] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242. USENIX, August 2003.

[30] Manigandan Radhakrishnan and Jon A. Solworth. Application security support in the operating system kernel. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)*, pages 201–211, Taipei, Taiwan, March 2006.

[31] Ronald Rivest, Adi Shamir, and L. Adleman. On digital signatures and public key cryptosystems. *Communications of the ACM (CACM)*, 21:120–126, 1978.

[32] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[33] Vipin Samar. Unified login with Pluggable Authentication Modules (PAM). In Clifford Neuman, editor, *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 1–10. ACM Press, 1996.

[34] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, December 2001. Revised April 2002.

[35] Jon A. Solworth and Robert H. Sloan. Decidable administrative controls based on security properties, 2004. Available at `http://www.rites.uic.edu/~solworth/kernelSec.html`.

[36] Jon A. Solworth and Robert H. Sloan. A layered design of discretionary access controls with decidable properties. In *Proc. IEEE Symp. Security and Privacy*, pages 56–67, 2004.

[37] Jon A. Solworth and Robert H. Sloan. Security property-based administrative controls. In *Proc. European Symp. Research in Computer Security (ESORICS)*, volume 3139 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2004.

[38] Jennifer G. Steiner, B. Clifford Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988.

[39] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[40] B. Swander, A. Huttunen, V. Volpe, and L. DiBurro. RFC 3948: UDP encapsulation of IPsec ESP packets, January 2005.

[41] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux Kernel. In *Proc. of the USENIX Security Symposium*, San Francisco, Ca., 2002.

[42] Tatu Ylonen. SSH—secure login connections over the internet. In *Proc. of the USENIX Security Symposium*, pages 37–42, San Jose, California, 1996.

# Hypervisor Support for Identifying Covertly Executing Binaries

Lionel Litty    H. Andrés Lagar-Cavilla
*Dept. of Computer Science*
*University of Toronto*
{llitty,andreslc}@cs.toronto.edu

David Lie
*Dept. of Elec. and Comp. Eng.*
*University of Toronto*
lie@eecg.toronto.edu

## Abstract

Hypervisors have been proposed as a security tool to defend against malware that subverts the OS kernel. However, hypervisors must deal with the *semantic gap* between the low-level information available to them and the high-level OS abstractions they need for analysis. To bridge this gap, systems have proposed making assumptions derived from the kernel source code or symbol information. Unfortunately, this information is *non-binding* – rootkits are not bound to uphold these assumptions and can escape detection by breaking them.

In this paper, we introduce *Patagonix*, a hypervisor-based system that detects and identifies covertly executing binaries without making assumptions about the OS kernel. Instead, Patagonix depends only on the processor hardware to detect code execution and on the binary format specifications of executables to identify code and verify code modifications. With this, Patagonix can provide trustworthy information about the binaries running on a system, as well as detect when a rootkit is hiding or tampering with executing code.

We have implemented a Patagonix prototype on the Xen 3.0.3 hypervisor. Because Patagonix makes no assumptions about the OS kernel, it can identify code from application and kernel binaries on both Linux and Windows XP. Patagonix introduces less than 3% overhead on most applications.

## 1 Introduction

Malicious software, otherwise known as *malware*, continues to be a serious problem in today's computing environment. Malware is becoming increasingly difficult to detect and remove because it commonly comes bundled with a *rootkit* [12], which abuses administrative privileges to hide the execution of malware binaries and their resource usage from the system administrator. Rootkits accomplish this by attacking the administrator's ability to obtain information about a system. For example, rootkits will subvert execution-reporting utilities, such as `ps` and `lsmod` on Linux systems and the `task manager` and `Process Explorer` [27] on Windows, which administrators rely on to query the operating system (OS) about running binaries and kernel modules. Rootkits may also subvert the OS kernel itself so that any queries to the kernel will receive a response that has been appropriately distorted by the rootkit. In this way, rootkits have been able to elude even the most experienced system administrators and sophisticated malware detection tools [11]. Even if the rootkit's presence is discovered, it is difficult to determine whether an attempted removal is successful or not, as the rootkit's ability to hide executing code enables it to trick the administrator into believing that it has been removed. As a result, best practice states that when a rootkit is even suspected to be present, the administrator must re-install the entire system from scratch to be sure that the rootkit is removed – a costly and undesirable solution. Trustworthy execution-reporting utilities, which would enable a system to detect hidden malware processes and determine if an attempted removal was successful or not, would save administrators a great deal of effort and reduce system downtime.

In this paper, we present Patagonix, a system that denies rootkits the ability to hide executing binaries from the system administrator. Patagonix does this by addressing two shortcomings of current execution-reporting utilities. First, these utilities all depend on the integrity of the kernel, both as a source of information and for protection against tampering. However, since rootkits can subvert the kernel, the trust that these utilities and the administrator invest in the kernel is misplaced. Second, these utilities do not verify the integrity of the binaries they report as executing. This shortcoming allows a rootkit to covertly execute code by injecting malicious code into a running binary or by tampering with the binary image on disk. Utilities that monitor binaries on disk, such as Tripwire [17], may detect tampering of on disk binaries, but will miss tampering of binaries once they are loaded in memory.

Unlike existing execution-reporting utilities, Patagonix does not depend on the OS. Instead, Patagonix uses a hypervisor, allowing it to retain its integrity even if the rootkit has compromised the OS kernel. The challenge to implementing an execution-reporting utility in a hypervisor is the *semantic gap* [6] between the information available to the hypervisor and the actual state of the system. Other work has bridged this gap by using and trusting information about the OS kernel, such as the kernel source code or kernel symbol information [3, 10, 13, 23, 25]. However, such information cannot be trusted because it is *non-binding* – the rootkit is not bound to maintain the semantics implied by source and symbol information, allowing it to escape detection. For example, if the hypervisor uses non-binding information about the format or location of kernel data structures, the rootkit may evade detection by adding fields to the data structures or moving the data structures to a memory location that is not being monitored. Similarly, assumptions about the code structure of the kernel can be exploited by a rootkit that modifies OS kernel execution to avoid code paths monitored by the hypervisor. Patagonix does not rely on any non-binding information about the OS kernel and relies only on the behavior of the hardware, which cannot be altered by malware.

Patagonix also verifies the integrity of all executing binaries before giving their identity to the administrator. Rather than verifying the contents of binaries on disk, Patagonix inspects the code as it executes in memory. As a result, Patagonix cannot be fooled by rootkits that avoid tampering with files on disk by injecting malicious code into binaries as they run. On the other hand, systems make modifications to code at run-time, causing it to differ from its image on disk when it is executed. Patagonix can differentiate legitimate modifications from malicious ones. The executing code is identified using a trusted external database that contains cryptographic hashes of binaries, such as the National Software Reference Library (NSRL) [20].

In this paper we make three main contributions:

- **Patagonix Prototype.** We have implemented a Patagonix prototype that leverages the capabilities of a hypervisor and the non-executable (NX) bit of the Memory Management Unit (MMU) to detect and identify all executing binaries regardless of the state of the OS kernel. Our prototype, built on the Xen 3.0.3 hypervisor [4], makes no assumptions about the OS kernel. As a result, with the exception of the binary format information, which differs from OS to OS, it can be used to neutralize rootkits on Windows XP, Linux 2.4 and Linux 2.6 OSs without modification.

- **Identity Oracles.** The semantic gap between the hypervisor and the OS requires special support to differentiate legitimate modifications made to running code by the OS from malicious ones made by a rootkit. To differentiate legitimate modifications from malicious tampering, we introduce the concept of an *identity oracle*, which when given a page of code in memory and a database of binaries, will either identify the binary from which the code page originated, or indicate that the code page is not from any of the binaries in the database. We have designed an oracle construction framework and implemented identity oracles for ELF binaries, PE binaries, the Linux kernel, the Windows XP kernel, and Windows driver interrupt handlers.

- **System Usage and Evaluation.** We present two complementary usage modes for Patagonix. In *reporting mode*, Patagonix serves as a trusted replacement for the standard execution-reporting utilities of an OS, allowing the administrator to see all executing processes even if hidden by a rootkit. This augments the administrator's ability to audit the state of the system during regular inspections and after an attempted rootkit removal. In *lie detection mode*, Patagonix compares the executing binaries reported by the OS with the executing binaries it identifies and reports any discrepancies to the administrator [10]. We tested Patagonix on 9 rootkits and found that it was able to identify code hidden by every one of them. In addition, our Patagonix prototype introduces less than 3% performance overhead on most applications.

We do not claim that Patagonix can detect all rootkits since Patagonix focuses on detecting covertly executing binaries – a rootkit that does not hide executing binaries, but only hides files and network connections, would not be detected. Fortunately, techniques to detect such rootkits, which do not depend on non-binding information, already exist. For example, using direct access to a raw disk image can detect hidden files [13] and a network-based intrusion detection system can detect hidden network connections. However, to the best of our knowledge, all techniques to detect hidden processes depend on non-binding information, making Patagonix useful in those circumstances.

In Section 2, we describe the problem with trusting non-binding information, the assumptions that Patagonix relies on, and the guarantees and limitations it has. Section 3 gives an overview of the Patagonix architecture, while Sections 4 and 5 detail our identity oracles and our prototype implementation. In Section 6 we describe the two usage modes of Patagonix: reporting and lie detection. Section 7 evaluates Patagonix's effectiveness at detecting covert processes and performance overhead. Section 8 discusses related work and we conclude in Section 9.

## 2 Security Model

### 2.1 Problem Description

Systems that monitor OS-level events from a hypervisor must wrestle with the semantic gap between the state of the OS and the information available to the hypervisor. Previous systems have bridged this gap using non-binding information derived from source code and symbol information, but acknowledge that in doing so they make themselves vulnerable to a rootkit that is aware of their monitoring technique [3, 10, 13, 23, 25]. For instance, if the hypervisor monitors the system call table by using location information derived from non-binding sources, the rootkit can evade detection by altering the kernel's system call dispatch handler to use a table placed at a different location, and filled with pointers to malicious system call handlers. The hypervisor-based monitor would continue to monitor the original, unchanged system call table, which is no longer being used by the kernel. Unfortunately, preventing this attack by simply disallowing modification of kernel code will cause false positives because kernels employ self-modifying code. Manipulating the dispatch handler is only one example; similar assumptions based on non-binding information about data types or function entry-points are equally prone to subversion. More sophisticated techniques take a systematic approach to analyzing the Linux kernel memory state for tampering by malware, but they require ad hoc rules written with expert knowledge [24] or source code annotations that provide only partial protection [25]. Further, all the aforementioned approaches use a sampling approach, creating a window of vulnerability that may be exploited by malware to remain undetected.

Patagonix securely addresses the semantic gap problem by avoiding reliance on non-binding information. Rather it relies only on information from the processor hardware about pages containing executing code. In addition, Patagonix detects and validates run-time code modification and ensures that they conform to the modifications permitted in the binary format specification. Finally, by utilizing the processor MMU hardware, Patagonix provides continuous monitoring and detection with very little overhead.

### 2.2 Assumptions and Guarantees

To provide security guarantees, Patagonix relies on two properties of the hypervisor. First, Patagonix assumes that the hypervisor will protect both itself and Patagonix from tampering by a rootkit that has subverted the OS kernel. This assumption is consistent with the guarantees that hypervisors aim to provide. Second, Patagonix relies on the hypervisor to provide a secure communication channel between it and the user. Patagonix uses this channel to inform the user of what binaries it detects are running. Because the hypervisor is the only principal with direct access to the hardware, this channel can be provided in a straightforward way by providing separate consoles for the OS and Patagonix.

Patagonix identifies executing binaries by the cryptographic hash of the executing code. To convey this information to the administrator in a useful way, these hashes must be mapped to the name of a file or application. Extracting this mapping from the disk image is not trustworthy since a rootkit can tamper with the disk. Instead, Patagonix relies on a trusted database to provide such a mapping. This database is assumed to contain the names of all legitimate software binaries that the administrator has installed on the machine and can also optionally contain mappings of known malicious binaries. Any executing binary that does not match one in the database is identified as "not present" and should be scrutinized by the administrator. Publicly available databases currently exist – for example, our prototype uses the NSRL [20]. We note that the labeling of binaries as legitimate or malicious is made available purely for the convenience of the administrator and is not used by Patagonix. History has shown that such labeling may be flawed – there have been many documented cases of trojaned, vulnerable, or patently malicious binaries being distributed by reputable entities [11]. Patagonix correctly handles situations where malware is executing on the OS because it was incorrectly labeled as legitimate in the database. For example, Patagonix can be used to confirm that the incorrectly labeled application is no longer executing after an attempted removal.

Even with malware in control of the OS, Patagonix guarantees that it is able to identify and report all executing binaries. Rootkits may try to hide malware binaries from the administrator by either appropriating the name of a legitimate application, or by trying to make it invisible. Patagonix prevents the former by using mappings from the trusted database. This also defeats any attempts to inject malicious code into legitimate binaries on disk or in memory since this will alter the contents of the code when it executes. If the rootkit tries to hide the execution of a binary by subverting the OS kernel or execution-reporting utilities, Patagonix will still identify and report the executing binary to the administrator since Patagonix monitors the processor hardware for executing code, not the OS kernel. With these guarantees, Patagonix can report the identities of all executing binaries to the user in reporting mode. Correspondingly, in lie detection mode, it can notify the administrator of any discrepancies between the code it detects and that reported by the OS.

## 2.3 Limitations

The goal of Patagonix is to provide a trustworthy alternative to traditional OS execution-reporting utilities, thus denying rootkits the ability to hide executing binaries from the administrator. However, detecting and preventing the exploitation of vulnerabilities is outside the scope of Patagonix. For example, Patagonix does not detect attacks that do not inject new code, but instead alter the control flow of an application, such as in a return-to-libc attack [32]. More generally, neither Patagonix nor traditional execution-reporting utilities prevent legitimate applications from taking malicious actions as a result of malicious inputs. For example, the attacker can cause a legitimate interpreter or a just-in-time (JIT) compiler to perform malicious actions by using it to run a malicious script. Despite this, Patagonix provides strong and useful guarantees. While Patagonix cannot tell if a script is malicious or not, it guarantees that the administrator will be aware of all executing interpreters and JITs.

Identifying and verifying the integrity of interpreters is the same as other binaries because all the machine level instructions that can be executed by the interpreter are known a priori. However, this is not the case for JITs because they dynamically generate and execute code whose content can be heavily dependent on the workload and run-time state. Thus, once Patagonix identifies a program as a JIT, it will ignore pages it observes executing in the JIT address space that are not present in the trusted database (JITs must always execute code from their binary before any dynamically generated code, so Patagonix is always able to identify the process first). While a rootkit may exploit this to inject arbitrary code into the JIT and escape any sandboxing enforced by the JIT, Patagonix's guarantees still hold because the rootkit will not be able to hide the execution of the JIT, nor can the rootkit cause Patagonix to misidentify the JIT as another application.

Finally, as mentioned earlier, Patagonix used in lie detection mode is not a generic rootkit detector: it focuses on rootkits that hide executing binaries.

## 3 System Architecture

## 3.1 Overview

The architecture of Patagonix is illustrated in Figure 1. The majority of Patagonix is implemented in the *Patagonix VM*, while a small amount of functionality that requires kernel mode privileges is implemented in the hypervisor. The *Monitored VM* contains the *Monitored OS* for which the administrator wants trustworthy binary execution information and the hypervisor protects Patagonix from tampering by the monitored VM. While implementing Patagonix entirely within the hypervisor may reduce performance overhead, splitting the functionality of Patagonix into hypervisor and VM components has the benefits of increased modularity, ease of portability to a different hypervisor, and a reduction on the size of the code being added to the security critical hypervisor. As we shall see in Section 7, the boundary crossings between the hypervisor and VM components of Patagonix have a minimal impact on overall performance.

The Patagonix VM contains three components. First, several identity oracles, one for each type of binary in the monitored VM, enable Patagonix to identify pages of code that are executed in the monitored VM. The identity oracles use cryptographic hashes of binaries from the trusted database to identify binaries executing in the Patagonix VM. Second, a *management console* implements the interface between the user and Patagonix. Finally, the *control logic* coordinates events between the management console, the oracles and the hypervisor component of Patagonix.

Only the identity oracles are OS-specific as one must be written for every binary format used by the OS in the monitored VM. All other components, which we collectively refer to as the *Patagonix Framework*, are OS agnostic.

## 3.2 Patagonix Framework

The Patagonix framework has three main responsibilities. First, the framework must detect when code is being executed in the monitored VM. Second, when code execution is detected, it invokes the identity oracles to identify the code and maintain a list of executing code. The identity oracles will either match the executing code to an entry in the trusted database, or will indicate that the identity of the code is not present in the database. Finally, the framework is responsible for conveying these results to the user in a way that is free of tampering by malware in the monitored VM.

Detecting code execution is performed by the Patagonix hypervisor component using the non-executable (NX) page table bit, which is available on all recent AMD and Intel x86 processors. When set on a virtual page, this bit causes the processor to trap into the hypervisor component whenever code is executed on that page. The hypervisor component then informs the control logic in the Patagonix VM by sending it a virtual interrupt.

Frequent traps into the hypervisor will hurt performance so Patagonix uses the processor to only inform it when either code is executed for the first time, or code it

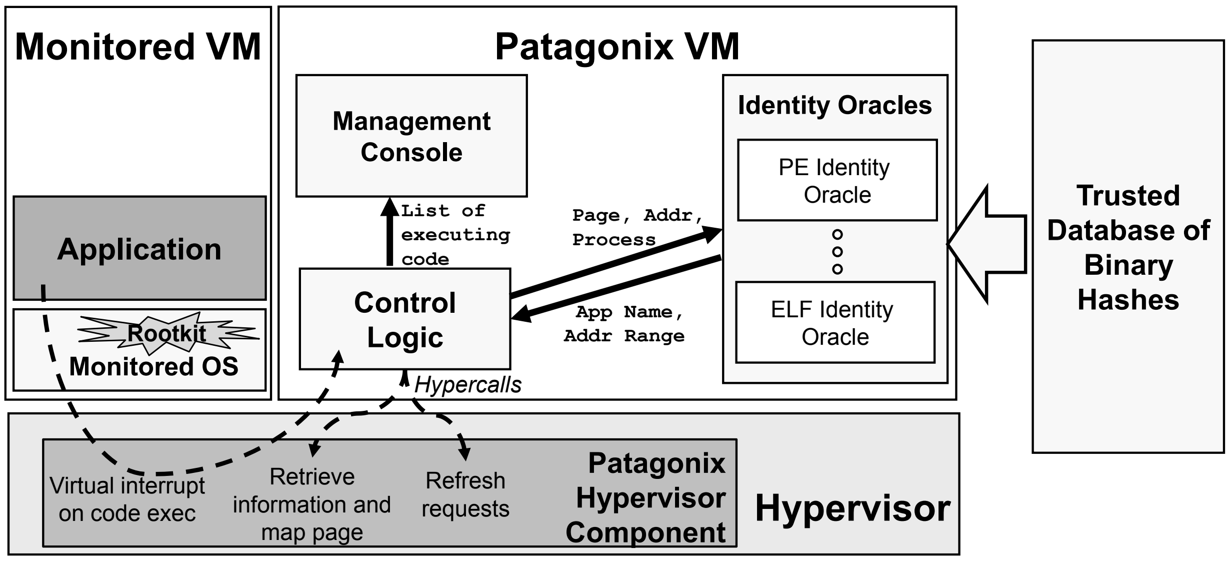


Figure 1: The Patagonix architecture.

has already identified changes and is executed. To identify code when it executes for the first time, the hypervisor component initially sets the NX-bit on all pages in the monitored VM so that it will receive a trap from the processor when a code page is executed. When it receives such a trap, the hypervisor component invokes the Patagonix VM to identify the code and then clears the NX-bit on the page, making it executable. At the same time, to detect if the identified code is subsequently modified, the hypervisor component makes the page read-only by clearing the writable bit in the page table. As long as the page remains unchanged, subsequent executions of code on that page do not cause a trap. If the identified code is modified, the processor will trap into the hypervisor, at which time the hypervisor component will make the page writable but non-executable again. If the modified code is executed, the hypervisor component will again receive a trap, at which point it will use the Patagonix VM to re-identify the code. To eliminate the possibility of a race where the rootkit alters the code page after it is identified, but before it is made executable, the monitored VM is paused while the Patagonix VM identifies the executing code. Setting executable or writable privileges on entire pages at a time is fairly straightforward. However, pages that contain mutable data and code require the ability to prevent writes to the code portions of the page and execution for the data portions of the page. While this can be implemented with additional hardware, we have been able to emulate such support in software. We defer the details of the solution to Section 5.2.

To identify code in memory, the identity oracles require the contents of the code page being executed, the virtual address at which the page is located, and the process the code comes from. The control logic retrieves this information via new *hypercalls*, which are hypervisor analogs of OS system calls we have added to Xen. The control logic then passes this information to each of the identity oracles, which either return the identity of the binary from which the code originated, or indicate that the identity of the originating binary is not in the trusted database. We note that Patagonix does not use OS process IDs to identify processes as these are controlled by the OS and can be subverted by a rootkit. Instead, Patagonix identifies a process by its virtual address space, which is an equivalent hardware proxy since by definition there is a one-to-one relationship between OS processes and address spaces. A process' address space is denoted by the base address of its page table hierarchy, which is maintained in a dedicated register on x86 processors.

Because the hardware only reports when code is executing, rather than when it is not going to be executed any more, the control logic records the most recent time it observed each binary execution and periodically instructs the hypervisor to perform a *refresh*, i.e., set all pages as non-executable. Code that is no longer executing will not trigger any more traps. Patagonix does not infer process termination by observing when a page table does not contain any valid mappings like Antfarm [14] because malware that controls the OS can toggle the page table bits between valid and invalid without actually removing the process from memory, thus circumventing this process termination heuristic.

The control logic uses the management console to securely report the list of observed executing binaries and times they were last observed executing. Because the hypervisor has control over the hardware, it is able to provide the management console in the Patagonix VM with an interface separate from that of the monitored VM, thus ensuring that the monitored VM cannot tamper with the output of the Patagonix VM.

## 3.3 Identity Oracles

Executable binaries are mapped from disk into memory by a *binary loader*, whose behavior is governed by the binary format that it loads. The task of the identity oracles is to use the information provided to them to reverse the transformations that the loader applies to binaries, and identify which binary in the trusted database (if any) the page of code being executed originates from.

Aside from the information provided to the oracles by the hypervisor component, the oracles also require information about the binaries in the database they are trying to match against. For example, information such as hashes of each individual code page in the file and information about relocations are required depending on the particular format of the binary. While current binary databases generally only contain hashes of binary files, additional information can be extracted from files on disk after they have been authenticated using the trusted database. Each oracle initially collects such information by searching the disk of the monitored VM for all executable binaries. The authenticity of an executable file is verified when its hash is found in the database, and the oracle can then proceed to extract additional information from the file. Patagonix needs to rescan the disk each time binaries are added, or alternatively, a program in the OS can be used to gather information about new binaries as they are introduced into the system. If an executable file is hidden from Patagonix by a rootkit, Patagonix will not have the necessary information to identify executing code from this binary and thus will not be able to match code originating from these binaries against entries in the database. As a result, such code will be identified as “not present”, thereby indicating to the administrator that a rootkit is likely on the system. In either case, access to the trusted database itself must be free of tampering by the rootkit. We implement our prototype database by combining hashes from the NSRL database, hashes from signed RPM packages and hashes computed from pristine binaries directly into the Patagonix VM image. Had the database been maintained remotely, it would need to be accessed over a secure, authenticated channel such as one offered by SSL.

Once the information about the binaries is acquired, the main challenge for the oracles is to reverse the transformations done by the loader without trusting information from the OS. Formally, each binary loader can be modeled as a function $L(B, S) = (\mathbf{M}, \mathbf{A})$, which maps a particular binary $B$, and the state of the OS at the binary load-time $S$, to a set of memory pages $\mathbf{M}$ and a set of addresses $\mathbf{A}$. $\mathbf{M}$ denotes the set of possible executable pages that the loader may transform the binary into and $\mathbf{A}$ denotes the possible virtual addresses at which the loader may place the transformed binary. The oracle for a particular binary format is a function $O_L(M, A, P) = \mathbf{B}$, which given a page $M$ detected as executing by the hypervisor, the virtual address of the executing code $A$, and the process it was executing in $P$, produces a set of binaries $\mathbf{B}$, from which the page could have originated. Since $M$ and $A$ are produced by the loader, they are elements of sets $\mathbf{M}$ and $\mathbf{A}$ respectively. One cannot implement $O_L$ by only relying on $S$, since a rootkit can subvert $S$. This inability to safely infer $S$ represents the semantic gap that the identity oracles bridge. Since we do not know $S$, $O_L$'s task can be generalized to searching the set $\mathbf{MA}'$ for the observed code page and address $(M, A)$, where $\mathbf{MA}'$ contains all code page/address combinations that the loader could have generated for all binaries and all legitimate OS states.

$\mathbf{MA}'$ can be very large, making the performance cost of a naïve search impractical. For example, in Windows, a code page can be mapped at $2^{20}$ possible locations (for a 32-bit address space when using 4KB pages) and its contents will be different for each of those possible locations. If applied to code pages in all binaries in an average Windows installation, this would result in an $\mathbf{MA}'$ several terabytes in size, which would be overly expensive to search. To reduce these costs, we exploit two characteristics that every binary format we have examined exhibits. The first is that these formats specify that code sections should be mapped to contiguous regions of memory. As a result, once the binary that occupies a memory region in a process is known, the oracle only needs to check that other code executing in the same region is the appropriate page in the same binary, eliminating the need to search $\mathbf{MA}'$ in these instances (in this case, binary can refer to a program binary or a dynamically linked library). Knowing the address where a binary is mapped also enables the oracle to reverse run-time modifications and derive the original code page, eliminating the need to store all versions of the page. To establish what binary occupies a region, the oracle exploits the second characteristic: binary executables have only a few entry-points (usually only one), which are executed before any other code in the binary. As a result, if code executes in a memory region where the oracle has not identified a binary before, the oracle only has to check for code at pages containing entry-points in $\mathbf{MA}'$. This reduces the search space, and also adds

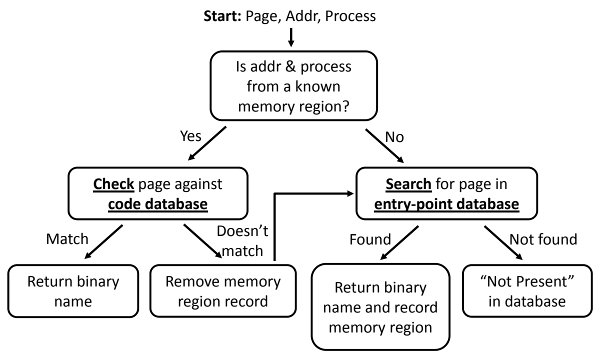

Start: Page, Addr, Process
Is addr & process from a known memory region?
Yes
No
Check page against code database
Search for page in entry-point database
Match
Doesn't match
Found
Not found
Return binary name
Remove memory region record
Return binary name and record memory region
"Not Present" in database

Figure 2: Identity Oracle framework. The functions and databases that are loader specific have been underlined.

a desirable security check since the oracle will identify code as "not present" if the malware tries to jump into a binary at any point other than a legitimate entry-point. We use these assumptions about binaries as hints to improve the performance of Patagonix. However, Patagonix does not trust these hints, so its security guarantees are not affected – tampering with the binaries that violates these assumptions will result in the tampered binary being identified as "not present".

Figure 2 illustrates our oracle construction framework. Four components in the framework are binary loader specific. The first is an *entry-point database*, which contains information on the entry-points of known binaries. This database is searched using an entry-point *search function*. The other two components are the *code database*, which contains information on the rest of the code sorted by binary, and the code *check function* which checks code against the code database. An oracle invocation begins with the control logic forwarding the page contents, faulting virtual address and process to the oracle. The oracle first checks whether the virtual address and process of the code are from a region where the binary is known. If not, then the binary has just started executing because no code has been observed executing at this location before. The oracle searches the entry-point database for a match to identify the binary. If a match is found, it records the binary's name and memory range it should occupy and returns the name of the binary. Otherwise, the oracle identifies the code as "not present" in the database.

If the address is from a memory region whose binary has been previously identified, then the oracle checks that the executing page is from the associated binary. If it is, the oracle returns the name of the binary. If it is not, then the binary no longer occupies that memory range in that process. The memory region record is removed and the oracle searches for the page in the entry-point database.

We have observed cases of related binaries containing identical code pages. If there have not been enough pages executed to uniquely identify the binary, the identity oracles return a list of candidate binaries until a unique page of code is executed. Should a page contain a mix of data and code, the oracles also return the sub-page range of the code.

## 4 Oracle Implementation

In this section, we describe the oracles we have constructed for various binary formats and their loaders. We find that while binary formats may differ, the operations performed by the loaders of these formats have similarities, allowing common techniques to be used across the oracles for different formats. We classify our oracles into two categories based on the type of binaries they identify. The first category consists of oracles for application code in Linux and Windows. We discuss support for the two main methods for dynamic code loading: position independent code and run-time code relocation, both of which are represented in the ELF and PE formats used by Linux and Windows respectively. The other category consists of kernel code in Linux and Windows. This code poses some extra challenges because both kernels contain self-modifying code. However, our oracles are able to verify that they are applied correctly. Finally, we finish this section with a discussion on the generality of our identity oracles.

### 4.1 Application Binary Oracles

**ELF Oracle.** The Executable and Linkable Format (ELF) [33] is used by Linux, as well as other OSs such as Solaris, IRIX and OpenBSD. An ELF file is divided into segments and contains a program header table that specifies the address at which each segment should be mapped into memory. ELF segments in the binary are identical to the segments that will be loaded in memory and no run-time modifications are required from the loader. Code in executable segments can either be relocatable, meaning it can be loaded at any address in memory, or non-relocatable, meaning that it must be loaded at a particular address. All references to absolute addresses in relocatable code go through indirection tables, which are filled in by the run-time linker. ELF shared libraries are typically relocatable, while executable binaries are typically non-relocatable.

Since ELF shared libraries use position independent code, both ELF libraries and ELF applications are mapped from disk into memory without any modifications, making this our simplest oracle. To populate the entry-point database for the ELF oracle, pages containing entry-points are placed in the database – all shared objects have an `init` subroutine that is run when the shared object is loaded and executables always begin execution in `_start`. To save space, the ELF oracle does not store the entire page contents in the database, but instead stores a cryptographic hash (SHA-256) of the page instead. The hashes are stored in a sorted list and the entry-point search function computes the hash of the page where code execution was detected and searches the entry-point database for a match.

The code database stores hashes of all pages for each binary in a two dimensional array that is indexed first by binary and second by page offset from the beginning of the binary. The check function uses the binary name attached to the memory region to compute the first index in a look up and the offset of the executing page from the start of the memory region to compute the second index. A hash of the executing page is then compared to the hash from the code database. Because SHA-256 is collision-resistant and difficult to invert, any tampering of the binary will result in the binary being identified as not present.

**PE Oracle.** The Portable Executable (PE) format [19] is used in all versions of Windows after Windows NT 3.1. Similar to ELF files, PE files have a header table that describes how sections in the file should be mapped in memory. However, code in PE files contains absolute addresses, and thus is not position independent. All PE files have an image base, which indicates the *preferred address* for loading the file. If an application needs to load two or more Dynamically Linked Libraries (DLL) that occupy overlapping preferred address regions, the OS must *relocate* one or more of the binaries. To do this, the absolute addresses in the executable must be adjusted by adding the offset between the preferred address and the actual address where the binary is loaded. This relocation operation is performed by the OS using the information stored in the binary header.

PE binaries pose two challenges. First, because the OS may adjust the absolute addresses in a binary, one cannot directly use page contents to identify code pages in the entry-point database. Instead, the PE oracle exploits the fact that the PE loader only relocates binaries by 4KB page offsets, meaning that the offset of the entry-point from the top of the page (i.e. the page-offset) is always the same. Thus, the entry-point database is indexed by the page-offset of the entry point and contains the locations of the absolute addresses in each candidate page, as well as a hash of its contents. The search function then searches the entry-point database for the page-offset of the faulting address to determine the binary.

In some cases, several binaries may have the same entry-point offset, so the search function must find the matching page within a set of more than one candidate pages. For each candidate, the search function undoes the absolute address adjustments made by the OS during relocation. This is accomplished by making a copy of the executed page and subtracting the relocation offset from each absolute address. This offset is the difference between the entry-point address of the executed page and the entry-point address of the candidate if it were mapped at its preferred address. A hash of the copy can then be compared against the hash of the candidate.

The second challenge is that some PE binaries have memory pages that contain both code and mutable data. For example, the Import Address Table (IAT), which is used to dynamically link DLLs against an application, is typically put in the code section by the Microsoft compiler. As a result, the search function only uses the portions of these pages that contain code to identify them, and will notify the control logic, which in turn will instruct the hypervisor to make only the identified portions of the pages executable. Naturally, the entry-point database entries for these pages must also contain information listing what portions of the page contain code.

The rest of the PE oracle is straightforward. The code database and check function are also similar to the ELF oracle except that they must undo any relocations before comparing the page contents and they must account for pages that only partially contain executable code. Thus, the code database also stores the preferred address with each binary, and the locations of all absolute addresses and sub-page code ranges (if necessary) with each page entry. To undo the relocations, the check function uses the actual address the binary was mapped in at, which is given by the start address of the memory region record, and then uses the same technique as the entry-point search function. In this way, the PE oracle provides the same guarantees as the ELF oracle.

## 4.2 Kernel Binary Oracles

**Linux Kernel Oracle.** The Linux kernel's code pages in memory are not always identical to their on-disk representation. Recent versions of the Linux kernel customize their binaries at run-time depending on the availability of more efficient instructions for the CPU the kernel is executing on. For example, the kernel will implement memory barriers with `LFENCE` and `MFENCE` instructions if running on newer x86 processors with SSE2 extensions. Altering these instructions at run-time allows a single kernel binary to be used on different CPUs. In addition, the Linux kernel can dynamically load and unload kernel modules at run-time.

The aspects of the Linux kernel that differentiate it from application code are self-modifying code and the ability to dynamically load modules. However, both of these can be handled with the techniques used in the PE oracle. In the Linux kernel, the locations of customizable instructions, the instructions they can be replaced with, and the conditions to permit replacement are stored in special sections of the kernel binary. Using this information, the search and check functions make a copy of the page, verify that the substitutions are legitimate, and then undo them by replacing them with the default on-disk instructions. The pages are then hashed and compared against the entries in the databases.

Linux kernel modules can be loaded at any location in memory and have both relocations and customizations that are adjusted at load-time. They also contain an initialization function that can serve as an entry-point for the module, making their loader very similar to that of a PE DLL. As a result, much like in the PE oracle, the Linux kernel oracle uses an entry-point database consisting of entry-point offsets. Once a kernel module is identified, the memory range it occupies is recorded.

**Windows Kernel Oracle** The Windows kernel exhibits behavior similar to the Linux kernel, where some of its code pages are customized at run-time by patching the kernel code. In addition, Windows also permits run-time loading of kernel modules and drivers.

Unlike the Linux kernel, the Windows kernel's replacements are not specified in the kernel binary, but are applied in an ad hoc fashion by various functions throughout the kernel. However, since these customizations are deterministic for a given hardware platform and occur early during boot, it is possible to record the customizations from a pristine kernel and use these to verify the customizations in the monitored VM. While this approach cannot guarantee completeness (for example, we do not know what replacements will take place for other hardware), we believe that a developer with more information about the Windows kernel customizations would be able to exhaustively enumerate the transformations the kernel performs at run-time. The Windows kernel oracle handles the run-time loading of drivers in exactly the same way as the Linux kernel oracle.

Both the Linux kernel oracle and the Windows kernel oracle provide the same guarantees as the ELF and PE oracles. While the PE oracle validates relocations by using the difference between the actual address and the preferred address, the kernel oracles perform an equivalent validation for run-time customizations by ensuring that modified instructions are replaced with legitimate substitutes.

**Windows Interrupt Handler Oracle.** To allow drivers to register interrupt service routines, the Windows kernel provides an *interrupt object* abstraction. To allow for driver portability, when such an interrupt object is initialized by the driver, 106 bytes of kernel-specific code is copied from an interrupt handling template into the object, and will be executed whenever an interrupt associated with the object occurs [28].

While this appears to be a form of dynamic code generation, it is actually very easy to write an oracle that identifies the Windows Interrupt Handler. The code is shorter than a page, so it can be efficiently identified and validated in its entirety with one oracle invocation. As a result, the Interrupt Handler oracle does not need a code database or check function. Furthermore, the code is exactly the same every time it is copied except for an 8 byte field that contains run-time parameters and absolute addresses, which is customized for each driver. As a result, no entry-point database exists for this oracle, and the search function simply performs a byte-by-byte comparison of the code starting at the faulting address with the 106 byte template. If there is a match, the code is identified as a Windows Interrupt Handler and only the 106 byte region is made executable and non-writable.

Our prototype oracle currently does not perform further checks on the 8 bytes that are modified dynamically by the kernel. This means that an attacker can arbitrarily modify these bytes. However, this is a small amount of memory, and these bytes are not contiguous. A more sophisticated oracle could also validate the contents of these bytes.

## 4.3 Discussion

To better understand the generality of the approaches we have employed for our prototype oracles, we examined descriptions of other common binary formats and loaders. We found that for application code, the main reason for run-time code modifications is to support the need to be able to dynamically load libraries at any base address. Nearly every binary format we examined, which included common formats such as the Mac OS X Mach-O format, the COFF format used by SysV, and a.out, uses either position independent code or rebasing – both of which we are able to handle.

Another interesting class of loaders are executable packers. They incorporate code into a compressed binary to decompress the code just before execution. As a result, the compressed binary needs to be unpacked first before the oracle gathers information from it. This extra step is conducted when Patagonix adds a packed binary to the code database. Our prototype currently only handles binaries that have been packed using the popular UPX [21]. To support additional packers, Patagonix only needs to be provided with an unpacker. For example, Patagonix could use PolyUnpack [26] to automatically support a large number of executable packers.

Finally, we observed two non-JIT binaries that dynamically generate code: `winlogon.exe`, which authenticates users, and the Windows Genuine Advantage application, which checks the Windows OS for evidence of piracy. No formal specification exists for the code generated by these applications and there is evidence that the code is generated to obfuscate self-integrity-checking operations. Without more information (like we had for the Windows interrupt handlers) or reverse engineering (which would violate the EULA), we cannot build an oracle that validates the legitimacy of the generated code. Thus, these binaries are treated as JITs – we can identify that they are executing, but do not examine other code pages in their address space.

## 5 Framework Implementation

We used the Xen 3.0.3 hypervisor as a basis for building our Patagonix prototype. When used in Hardware Virtual Machine (HVM) mode, Xen utilizes virtualization support in x86 processors to run unmodified operating systems, including both Linux and Windows. With the exception of our emulated sub-page privileges support, our implementation of Patagonix can run on both AMD and Intel processors. In implementing Patagonix, we found that while the MMU provides a way to efficiently detect code execution, care needs to be taken to ensure that all code execution in the monitored VM is detected. Another shortcoming of the processor support was the inability to allow or deny execution or write pages at a sub-page granularity. Finally, we discuss a performance optimization that reduces the number of Patagonix VM invocations the hypervisor must make.

### 5.1 Detecting Code Execution

The non-executable permission bit was primarily implemented to allow an OS to prevent unauthorized code execution. When this mechanism is virtualized, there are two issues that must be taken into account to ensure that all instances of new code execution are detected by the hypervisor.

The first issue arises from the fact that page permission bits apply to a virtual page mapping and not to a physical page. Since there can be more than one virtual mapping for a physical page, our hypervisor modifications must ensure that there cannot be writable and executable mappings of a physical page simultaneously. Otherwise, the rootkit could use one mapping to modify the page and the other to execute it. We accomplish this by leveraging Xen's frame map, which maintains a count of the number of mappings of each physical page. Whenever a page changes from writable to executable or vice versa, Xen consults the count in the frame map to see if any other virtual mappings need to be updated appropriately. Xen's frame map only maintains a count of the number of mappings, and is not a reverse frame-map; as a result, we must walk the page tables to find and change all other mappings.

This issue could also be fixed by upcoming nested-page table (NPT) support, which provides full hardware virtualization support for page tables. NPTs add a shadow page table, which allows the hypervisor to specify a second translation between the guest physical frame numbers and the actual machine frame numbers. With this, the hypervisor could simply control the permissions for the machine frames, removing the need to track the number of guest virtual mappings for each physical page. To be notified when new code is executed, Patagonix marks pages as non-executable in the shadow page table, and then makes them executable after they have been identified. We do note that in doing this, Patagonix will negate one of the possible advantages of NPTs, which is to allow superpage mapping of a contiguous set of guest physical frames with a single NPT entry.

The second issue stems from the fact that the virtual Direct Memory Access (DMA) unit in Xen runs in a separate protection domain (the privileged `domain0`) and thus is not constrained by the page access restrictions placed on the rest of the monitored VM. Malware that is aware of this could abuse the virtualized DMA to modify memory pages that have been marked as executable and read-only. To make sure that memory content was always checked before being executed, we modified the emulated DMA devices to inform the hypervisor when they write to any pages. If any of these pages are marked as executable, Xen makes these pages non-executable again.

### 5.2 Sub-page support

Sub-page permissions are necessary when a memory page contains a mix of identified code and mutable data: the code must be made non-writable, and the data must be made non-executable. Ideally, sub-page support would be provided in hardware using a scheme such as Mondrian memory [35] or Transmeta's Crusoe processor [8]. However, because such support is not available on x86 processors, we devised a method to emulate this support based loosely on a technique that Van Oorschot et al. used to circumvent code tampering detection [34]. The technique takes advantage of the separate Translation Lookaside Buffers (TLB) for instructions (ITLB) and data (DTLB) present in x86 processors.

Our solution maps an execute-safe version of the page to a virtual address for instructions, and the original to the same virtual address for data. The execute-safe version is a copy of the mixed page where the data sections have been made non-executable by replacing them with trap instructions. A mapping to this version is loaded into the ITLB by temporarily setting the shadow page table entry to be executable, pointing it to the execute-safe version and executing a single instruction from that page. After that, the shadow page table entry is switched back to the original page and made writable and non-executable. This emulates the sub-page permission control we require since any attempt to execute at an address from the data regions will go through the ITLB and result in a trap, and any modifications to the code region will go through the DTLB and will not be applied to the page that instructions are being fetched from. To ensure that the execute-safe page is not accidentally loaded into the DTLB by an unintended load or store while setting up the TLBs, Patagonix disables interrupts for the monitored VM during this operation.

The emulation has some drawbacks over native hardware support. First, the emulation does not trap into the hypervisor when a write is attempted to a code region. Such functionality would be needed to deal with run-time modifications to a mixed page, but we have not found this necessary in practice. Second, this TLB manipulation needs to be undertaken every time to correctly load the ITLB mapping for this page, ITLB misses for such pages are transformed into page faults that require two traps into the hypervisor. Finally, this functionality cannot be emulated on Intel processors because, at the time of writing, Intel processors flush both TLBs on every crossing between the hypervisor and the VM.

## 5.3 Performance Optimizations

The dominant source of overhead in Patagonix is the page faults that occur when the monitored VM executes pages marked non-executable by Patagonix and the subsequent Patagonix VM invocation to identify the newly executing code. Some of these page faults are unnecessary because the executing code is on a physical page that has already been identified when it was executed in another process. Thus, we added an optimization that avoids the extra page fault and Patagonix VM invocation for pages whose identities are already known. This is accomplished by maintaining a list of physical pages that have been identified and whose virtual mappings are all executable and non-writable. When the monitored VM attempts to map such a page as executable in a new process, Patagonix preemptively makes the new mapping executable and non-writable.

The hypervisor must log each time this optimization is applied for two reasons. One reason is because this information is required to maintain the consistency of the memory region information for the oracles. The second reason is that this information is required by the Patagonix VM to maintain an accurate record of when pages from each binary were observed executing. To avoid extra domain crossings but keep the Patagonix VM's view of the monitored VM current, this log is read by the Patagonix VM whenever it is invoked by the hypervisor to identify a page, whenever it requests the hypervisor to perform a refresh and whenever the user requests a list of executed binaries through the management console. As a result, this optimization has no effect on how current the Patagonix VM's information on executing binaries is, and thus has no impact on the security guarantees of Patagonix.

## 6 Usage

Patagonix has two usage modes. In *reporting mode*, Patagonix provides trustworthy execution-reporting information and is functionally similar to utilities such as `ps`, `lsmod` and the `task manager`. This gives the system administrator a trustworthy alternative information source when evaluating if their system has processes hidden by a rootkit, or whether an attempted rootkit removal has been successful. In *lie detection mode*, Patagonix compares the list of executing binaries reported by the monitored OS with what it detects is executing. Differences mean that the OS is lying and indicate that a rootkit is present on the system.

When in reporting mode, Patagonix displays a list of all executing binaries on the management console. This is semantically similar to the list displayed by utilities such as `top` or the `task manager`. Patagonix also displays the times they were last observed executing. The administrator can also use Patagonix to terminate or suspend the execution of all instances of a binary by issuing commands to the management console, creating a trustworthy version of the UNIX `kill` utility. To terminate a binary, Patagonix sets all pages of that binary to non-executable. When an execution fault occurs on one of the code pages, Patagonix replaces the instruction at the faulting address with an illegal instruction. This makes it appear to the monitored OS that the binary tried to execute an illegal instruction, causing the monitored OS to terminate it. Suspending execution is achieved by replacing the code with an empty loop instead of replacing it with an illegal instruction. Thus, the binary is still executing from the OS' point of view, yet no code from the actual binary is being executed. A more efficient, but OS-specific implementation could inject code that causes the application to sleep.

In lie detection mode, Patagonix compares execution information reported by the monitored OS with its own list of executing binaries. Patagonix obtains execution information from the monitored OS via an agent in the monitored VM. The agent is a program that queries the monitored OS via standard interfaces to obtain a list of executing processes. Previous systems that performed lie detection in this way can suffer from false positives

| Target OS | Rootkits |
|---|---|
| Linux 2.4 | Adore, Adore-ng, Knark, Synapsys |
| Linux 2.6 | Adore-ng-2.6, Enyelkm |
| Windows XP | Fu, Hacker Defender, Vanquish |

Table 1: Rootkits detected by Patagonix. In reporting mode, Patagonix is able to identify processes hidden by these rootkits and/or detect tampering of processes by these rootkits. In lie detection mode, Patagonix detects that the OS is under reporting the binaries that are running.

due to asynchrony between the measurement of running processes taken from within the monitored OS and the measurement taken from the hypervisor – a new process may begin executing and be detected by the hypervisor before the OS has had a chance to update the information it exports to the agent [10,13]. To avoid this, Patagonix's agent registers a function with the OS kernel that synchronously informs Patagonix of process creation via a hypercall. Both Linux and Windows provide facilities for this.

Patagonix's lie detection detects both OS under-reporting (hiding executing binaries) and over-reporting (reporting binaries that are not actually executing). Usually, rootkits under-report to hide the execution of malicious binaries, but over-reporting could also be used maliciously. For example, a rootkit may wish to lead the administrator to believe that a critical program (such as an anti-virus scanner) is still running when it is not. Over-reporting requires the administrator to specify a threshold which dictates how long Patagonix will allow a binary that is reported as executing by the OS to be not observed running any code before declaring it as being over-reported.

## 7 Evaluation

We evaluate two aspects of Patagonix: its effectiveness at detecting and identifying hidden processes and rootkits and the performance overheads introduced by adding Patagonix to the hypervisor.

All experiments were carried out on a machine with an AMD Athlon 64 X2 Dual Core 3800+ processor running at 2GHz, with 2GB of RAM. We used the Xen 3.0.3 hypervisor and allocated 512MB of RAM to the monitored VM and 1GB of RAM to the `domain 0` VM, which also doubles as the Patagonix VM. Unless stated otherwise, the monitored VMs contain either Windows XP SP2 or Fedora Core 5 with a 2.6.19 Linux kernel.

### 7.1 Effectiveness

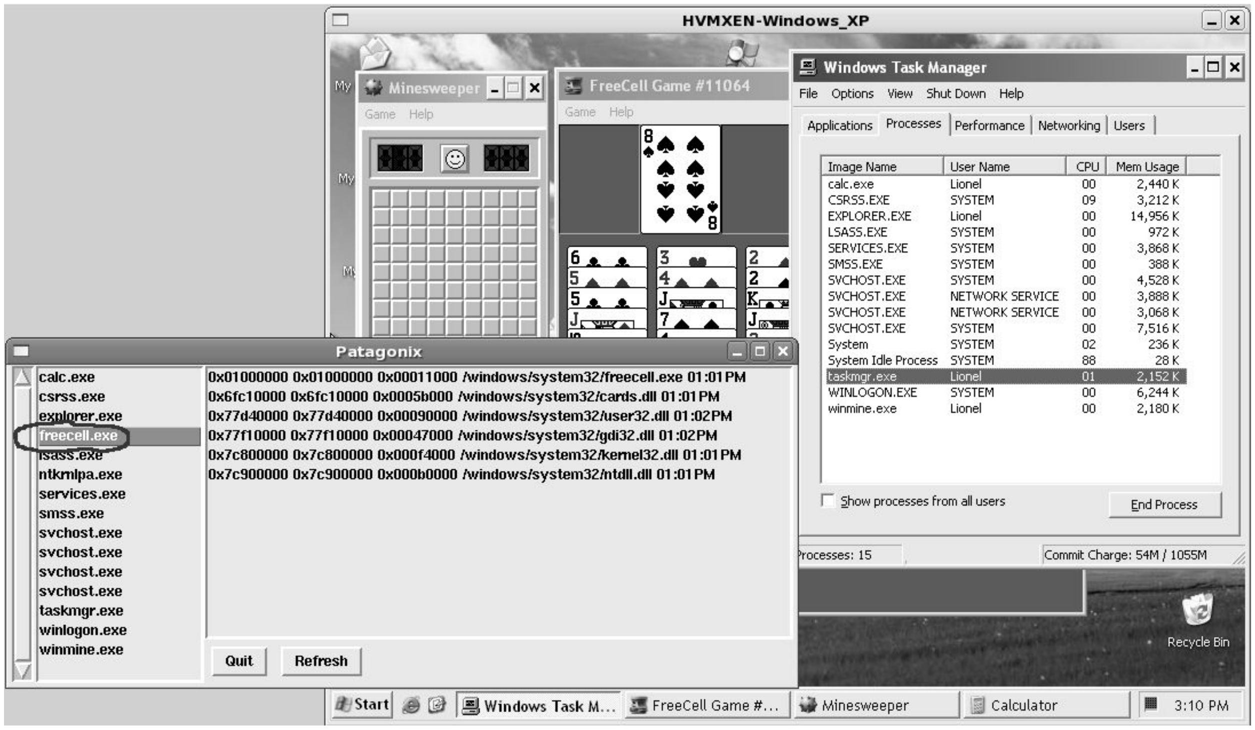
To evaluate the effectiveness of Patagonix at identifying covertly executing binaries, we used Patagonix to monitor VMs containing the nine rootkits listed in Table 1. These rootkits target the Windows kernel and Linux kernel versions 2.4 and 2.6. For this experiment, they were installed in VMs running Windows XP SP2, version 2.4.35.4 of the Linux kernel, and version 2.6.14.7 of the Linux kernel (The rootkits that targeted Linux 2.6 kernels did not work with version 2.6.19 of the kernel). We evaluated Patagonix in both reporting and lie detection mode.

First, we ran Patagonix on monitored VMs that have been infected with the rootkits. Each rootkit (except Vanquish) was configured to hide a process on the monitored OS: an instance of `Freecell` on Windows and an instance of `top` on Linux. We then verified that the hidden processes were not visible to the standard execution-reporting utilities on the respective OSs. In reporting mode, Patagonix was able to neutralize all the rootkits and report the execution of the covert code to the administrator, as illustrated in Figure 3. Likewise, in lie detection mode Patagonix is able to detect the tampering performed by each of the rootkits without fail. The Vanquish rootkit does not hide processes like the other rootkits. Instead, it tampers with applications by injecting code into the address space of executing processes. In these cases, the executing code of the tampered binaries is correctly identified as "not present" since it no longer matches any binary in the database. This warning should be interpreted as a likely rootkit infection by the administrator since the only other cause would be a missing binary in the trusted database.

Second, we ran Patagonix on VMs that did not have any rootkits installed to see if Patagonix reports any false positives. We exercise the VMs using the various application and microbenchmarks described in the following sections. During these tests, all executing code was correctly identified. When run in lie detection mode on an uninfected VM, Patagonix reported no discrepancies between the processes reported by the monitored OS and that detected by Patagonix.

### 7.2 Microbenchmark

To understand the overheads introduced by Patagonix, we devised *chain*, a microbenchmark that touches a new page of code on every instruction by chaining together a series of jumps, each targeting the beginning of the next page. Chain represents the worst case scenario for Patagonix: every instruction requires Patagonix to identify the new page of executable code. We instrumented our prototype to break down the page identification process



```
calc.exe
csrss.exe
explorer.exe
freecell.exe
lsass.exe
ntkrnlpa.exe
services.exe
smss.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
taskmgr.exe
winlogon.exe
winmine.exe
```

```
0x01000000 0x01000000 0x00011000 /windows/system32/freecell.exe 01:01PM
0x6fc10000 0x6fc10000 0x0005b000 /windows/system32/cards.dll 01:01PM
0x77d40000 0x77d40000 0x00090000 /windows/system32/user32.dll 01:02PM
0x77f10000 0x77f10000 0x00047000 /windows/system32/gdi32.dll 01:02PM
0x7c800000 0x7c800000 0x000f4000 /windows/system32/kernel32.dll 01:01PM
0x7c900000 0x7c900000 0x000b0000 /windows/system32/ntdll.dll 01:01PM
```

| Image Name | User Name | CPU | Mem Usage |
|---|---|---|---|
| calc.exe | Lionel | 00 | 2,440 K |
| CSRSS.EXE | SYSTEM | 09 | 3,212 K |
| EXPLORER.EXE | Lionel | 00 | 14,956 K |
| LSASS.EXE | SYSTEM | 00 | 972 K |
| SERVICES.EXE | SYSTEM | 00 | 3,868 K |
| SMSS.EXE | SYSTEM | 00 | 388 K |
| SVCHOST.EXE | SYSTEM | 00 | 4,528 K |
| SVCHOST.EXE | NETWORK SERVICE | 00 | 3,888 K |
| SVCHOST.EXE | NETWORK SERVICE | 00 | 3,068 K |
| SVCHOST.EXE | SYSTEM | 00 | 7,516 K |
| System | SYSTEM | 02 | 236 K |
| System Idle Process | SYSTEM | 88 | 28 K |
| taskmgr.exe | Lionel | 01 | 2,152 K |
| WINLOGON.EXE | SYSTEM | 00 | 6,244 K |
| winmine.exe | Lionel | 00 | 2,180 K |

Processes: 15 Commit Charge: 54M / 1055M

Figure 3: Output of both Patagonix and the Task Manager when the FU rootkit is used to hide `freecell.exe`. Patagonix identifies all processes including `freecell.exe`, while the Task Manager does not display the hidden process. Patagonix identifies "System" as `ntkrnlpa.exe`, the name of the Windows XP kernel binary.

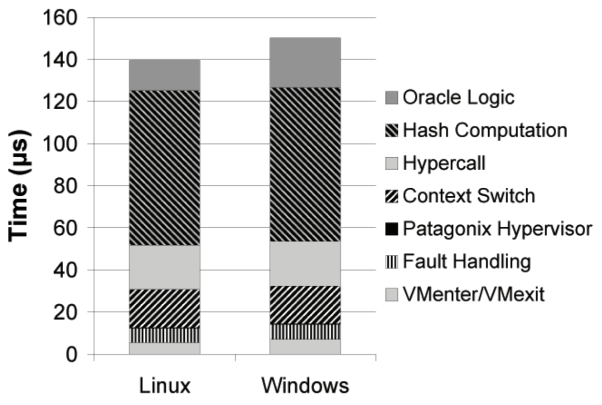


Figure 4: Execution time for various components of the identification operation. The total height of the bars represents the average time required to identify the origin of an executing code page.

into its different components. Figure 4 details the overhead incurred when identifying one page of code; the values presented are the average of 10,000 Patagonix invocations, and the standard deviations for each component were consistently less than 5% of the average.

When reaching a new page of code, a page fault is triggered by the MMU. This results in an unavoidable hardware cost due to the VMexit and VMenter operations in and out of the hypervisor. After a VMexit, a software page fault handling cost is incurred that is specific to Xen's shadow page table implementation; we expect it to change with other hypervisor implementations. The Patagonix's hypervisor code is then executed; running this code is extremely brief (approximately 0.3$\mu$s), attesting to its minimal impact on the hypervisor. This code triggers a context switch into the Patagonix VM, where a hypercall is executed to retrieve the executing page information. These two operations cost a total of 40$\mu$s, but enable 2080 out of a total 3544 lines of code to be implemented in the Patagonix VM instead of the hypervisor. The hash computation necessary for all oracles accounts for 73$\mu$s, nearly half of the page identification time. As expected, the PE oracle logic takes slightly more time than the ELF oracle logic. We note that the case in which the PE search function has to match an entry-point page against several candidates will be more expensive, as each candidate binary requires a hash computation; we have observed times as high as 538$\mu$s. Fortunately, this only happens very rarely and the search is only performed once per binary mapped in memory.

| Benchmark | Linux (%) | WinXP (%) | WinXP-hw (%) |
|---|---|---|---|
| Apache Build | 1.68 | 2.62 | 1.99 |
| Boot | 2.05 | 30.39 | 10.63 |
| SPECINT 2006 | 0.03 | 2.32 | 0.25 |
| perlbench | 2.06 | 23.01 | 1.42 |
| gcc | 13.75 | 12.43 | 3.48 |

Table 2: Application benchmark results. Results are the average of ten runs and are given in percent overhead over vanilla Xen. All standard deviations were less than 3% of the mean. WinXP-hw is estimated performance with hardware support for sub-page permissions.

## 7.3 Application Benchmarks

Since Patagonix is only invoked when code is executed for the first time, we expect this to coincide with page faults that load code from the disk. Because disk operations are expensive to begin with, we expect Patagonix overhead to be minimal in practice. To confirm this, we ran several application benchmarks in both the Linux and Windows VMs in our prototype. Computationally intensive applications are represented by the benchmarks from the SPECINT 2006 suite. For workloads with larger code footprints, we also measured the time Patagonix takes to boot Windows and Linux, as well as to build Apache. We compare the execution time for each benchmark against a vanilla Xen system running the same benchmark on the same monitored VM and report the overheads in Table 2. Since the PE oracle uses sub-page emulation, we also ran benchmarks without the emulation and sub-page checks (WinXP-hw column) to approximate what the performance might be if hardware support were available.

We report the SPECINT benchmarks as an aggregate because overheads for all benchmarks where less than 3% for the three configurations except for `gcc` and `perlbench`, whose performance we report separately. The Windows boot and `gcc` have large code footprints in comparison to their execution time: Windows initializes several services, drivers and interrupt handlers during boot, while SPEC drives `gcc` with a set of tests that exercises a large number of code paths. `perlbench` does not experience high overhead except in the WinXP configuration because it spends a high portion of its time running code on mixed code/data pages, motivating architectural support for sub-pages in such cases. As expected, the overhead for all other benchmarks is low. This is because their code footprint is small relative to their execution time.

Finally, the Patagonix VM needs to request periodic refreshes from the hypervisor. A shorter refresh interval means more accurate information about when a process was last observed executing, but also incurs more overhead. Figure 5 plots the additional overhead the Apache build benchmark in Linux experiences for various refresh periods, as well as the number of executable pages that are invalidated (set non-executable) each time. More frequent refreshes mean less time for the application to execute various pages, resulting in fewer invalidations.

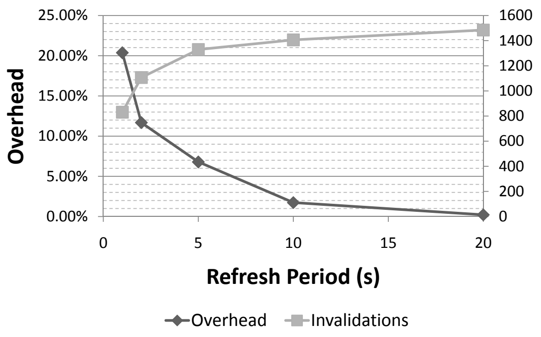


Figure 5: Overhead and Invalidations vs. Refresh Period. Apache Build on Linux. Averages of five runs with standard deviations below 2% of the average.

## 8 Related Work

The problems associated with the semantic gap between the hypervisor and guest VMs were first identified in a seminal paper by Chen and Noble [6]. Since then, there have been several attempts to bridge this gap using non-binding information derived from source code and symbol information. For example, Livewire [10], Copilot [23] and SBCFI [25] rely on symbol information in kernel binary or `System.map` file, while Asrigo et al. [3] and VMWatcher [13] rely on information derived from kernel source code. Because they make assumptions based on non-binding information, they are all prone to evasion by a rootkit that breaks those assumptions. Patagonix does not rely on any non-binding information.

The principle of lie detection – comparing two views of the same data for discrepancies – has been used in the literature. For example, Rootkit Revealer [7] and Strider GhostBuster [5] compare high-level and low-level views of the same system information. However, since both views are still derived from within the infected system, a thorough rootkit can make both high-level and low-level views agree, thus eluding these systems. Like Patagonix, other systems compare views taken from both within (i.e. in-the-box) the infected system, and outside (out-of-the-box) the infected systems. For example, both Livewire [10] and VMWatcher [13] compare views of executing processes derived from the VMM with those gathered from within the monitored system. However, unlike Patagonix, these systems do not deal with asynchrony between the measurement times of the in-the-box and out-of-the-box views and will thus suffer from false positives. Lycosid [15] also does lie detection by counting the number of address spaces in a VM. However, because Lycosid does not identify which binaries the processes are executing and the hypervisor's measurements contain noise, it can only probabilistically detect when the number of address spaces does not match the number of processes reported by the OS. Because Patagonix identifies processes and registers callbacks with the OS, Patagonix is able to both precisely detect hidden processes, as well as identify which process is being hidden.

Like Patagonix, remote attestation systems also must identify and report executing binaries on a system. In addition, they may also report the integrity of the data in a system, and are often used to report this information to a remote party instead of the system administrator. However, these systems in general assume a weaker attack model since they in general rely on the integrity of the OS. For example, IMA [29], implements such functionality directly in the OS kernel, and thus depends on the integrity of the OS kernel to report correct results. An alternative is Terra [9] which performs attestation in a hypervisor. Terra attests the identity of the virtual disk used to initialize a "closed box" to a remote party. Closed boxes are VMs that are fully managed by a third party and usually cannot be extended in any significant way. Since Patagonix allows the monitored OS to be arbitrarily extended as long as the hashes of any new legitimate code are in the trusted database. A combination of Patagonix and Terra's abilities could enable support for attestation of open, extensible systems as well as individual programs executing in these systems.

Hypervisors have long been used as a means for implementing a secure trusted computing base, with which untrusted OS images could be made secure [16, 31]. While our prototype was implemented in the Xen hypervisor [4], the functionality required from the hypervisor is generic enough to allow Patagonix to be implemented on any virtualization system. To explore this point, we have obtained a source code license for VMware Workstation and are currently working on a port of Patagonix. We have found that VMware-specific functionality, such as its page table entry caching [2] and dynamic code translation [1], have not impeded the necessary functionality from being added.

Finally, Patagonix uses or extends ideas presented in other work. Patagonix is based on our earlier work called Manitou, which also uses hashes to identify running applications from a hypervisor [18]. However, Manitou is only able to identify applications for Linux guest OSs, making its treatment of the problem overly simplistic. It also does not perform synchronous lie detection. Independent to our work and using a similar low-level mechanism to detect code execution, SecVisor [31] restricts what code can be executed by a modified Linux kernel. SecVisor focuses solely on code that is executed in kernel mode. It uses a custom-made hypervisor, showing that execution control can be achieved with a small TCB. In contrast, Patagonix provides comprehensive guarantees for unmodified Linux and Windows OSs as well as the applications they execute, and demonstrates that these guarantees can be obtained by small extensions to a general-purpose hypervisor. Other projects have manipulated the page tables used by the X86 MMU. For example, the PaX project [22] proposes manipulating these page tables to emulate the NX bit on older CPU that do no have hardware support for the feature. Finally, computer forensics experts [30] have demonstrated that PE binaries can be reconstructed by analyzing memory dumps. The PE identity oracle described in this paper uses similar techniques to identify binaries online.

## 9 Conclusions

Current OSs are vulnerable to subversion by rootkit and thus cannot be relied upon to provide trustworthy information about what code is executing on a system. Patagonix solves this problem by using the processor MMU to detect executing code from a hypervisor. It then uses identity oracles, which leverage information from the binary format specifications and loaders to identify the executing code. In this way, Patagonix is able to bridge the semantic gap between the hypervisor and the OS without having to trust non-binding information, which is vulnerable to subversion by the rootkit. We have found that binary formats across different OSs have similarities, enabling the creation of a universal oracle construction framework and the use of common techniques across various binary formats. Aside from the binary-specific formats, the Patagonix framework does not use any information about the OS, allowing the same framework to be used on diverse OSs such as Windows XP, Linux 2.4 and Linux 2.6, without any modification. Through the combined use of writable and non-executable page table bits, Patagonix is only invoked when code is executed for the first time, and as a result, has a modest performance overhead of less than 3% on most applications.

## Acknowledgements

This paper was greatly improved by comments from Tom Hart, Ian Sin, Jesse Pool, Lee Chew, James Huang, Stan Kvasov, Niraj Tolia and Ashvin Goel. We would also like to thank the anonymous reviewers for their helpful suggestions. This work was supported in part by an NSERC Discovery Grant and a MITACS Grant.

## References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, Oct. 2006.

[2] O. Agesen and P. Subrahmanyam. Method and system for performing virtual to physical address translations in a virtual machine. U.S. Patent 7069413, Dec. 2006.

[3] K. Asrigo, L. Litty, and D. Lie. Using VMM-based sensors to monitor honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 13–23, June 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Oct. 2003.

[5] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks (DSN)*, pages 368–377, Apr. 2005.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[7] B. Cogswell and M. Russinovich. RootkitRevealer v1.71, Nov. 2006.

[8] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, Mar. 2003.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, Oct. 2003.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, Feb. 2003.

[11] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th USENIX Security Symposium*, pages 77–92, Aug. 2006.

[12] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005.

[13] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, Oct. 2007.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 Annual Usenix Technical Conference*, pages 1–14, May 2006.

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, pages 91–100, Mar. 2008.

[16] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, 1990.

[17] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[18] L. Litty and D. Lie. Manitou: A layer-below approach to fighting malware. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 6–11, Oct. 2006.

[19] Microsoft. Visual Studio, Microsoft Portable Executable and Common Object File Format specification, May 2006. Rev. 8.0.

[20] NIST. National software reference library, 2008. `http://www.nsrl.nist.gov/`.

[21] M. Oberhumer, L. Molnár, and J. Reiser, 2008. `http://upx.sourceforge.net/`.

[22] PaX, 2008. `http://pax.grsecurity.net/`.

[23] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Aug. 2004.

[24] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, pages 289–304, July 2006.

[25] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, Oct. 2007.

[26] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 289–300, Dec. 2006.

[27] M. Russinovich. Process Explorer, 2007. `http://technet.microsoft.com/sysinternals/bb896653.aspx`.

[28] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 2005.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, Aug. 2004.

[30] A. Schuster. Reconstructing a binary, Apr. 2006. `http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html`.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[32] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–61, Oct. 2007.

[33] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) specification, May 1995. V1.2.

[34] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, Apr.-June 2005.

[35] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, Oct. 2002.

# Selective Versioning in a Secure Disk System

Swaminathan Sundararaman
*Stony Brook University*

Gopalan Sivathanu
*Stony Brook University*

Erez Zadok
*Stony Brook University*

## Abstract

Making vital disk data recoverable even in the event of OS compromises has become a necessity, in view of the increased prevalence of OS vulnerability exploits over the recent years. We present the design and implementation of a secure disk system, SVSDS, that performs selective, flexible, and transparent versioning of stored data, at the disk-level. In addition to versioning, SVSDS actively enforces constraints to protect executables and system log files. Most existing versioning solutions that operate at the disk-level are unaware of the higher-level abstractions of data, and hence are not customizable. We evolve a hybrid solution that combines the advantages of disk-level and file-system—level versioning systems thereby ensuring security, while at the same time allowing flexible policies. We implemented and evaluated a software-level prototype of SVSDS in the Linux kernel and it shows that the space and performance overheads associated with selective versioning at the disk level are minimal.

## 1 Introduction

Protecting disk data against malicious damage is one of the key requirements in computer systems security. Stored data is one the most valuable assets for most organizations and damage to such data often results in irrecoverable loss of money and man power. In today's computer systems, vulnerabilities in the OS are not uncommon. OS attacks through root kits, buffer overflows, or malware cause serious threat to critical applications and data. In spite of this, security policies and mechanisms are built at the OS level in most of today's computer systems. This results in wide-scale system compromise when an OS vulnerability is exploited, making the entire disk data open to attack.

To protect disk data even in the event of OS compromises, security mechanisms have to exist at a layer below the OS, such as the disk firmware. These mechanisms must not be overridable even by the highest privileged OS user, so that even if a malicious attacker gains OS root privileges, disk data would be protected.

Building security mechanisms at the disk-level comes with a key problem: traditional disk systems lack higher-level semantic knowledge and hence cannot implement flexible policies. For example, today's disk systems cannot differentiate between data and meta-data blocks or even identify whether a particular disk block is being used or is free. Disks have no knowledge of higher-level abstractions such as files or directories and hence are constrained in providing customized policies. This general problem of lack of information at the lower layers of the system is commonly referred to as the "information-gap" in the storage stack. Several existing works aim at bridging this information-gap [4, 11, 16, 18].

In this paper, we present the design and implementation of SVSDS, a secure disk system that transparently performs selective versioning of key data at the disk-level. By preserving older versions of data, SVSDS provides a window of time where data damaged by malicious attacks can be recovered through a secure administrative interface. In addition to this, SVSDS enforces two key constraints: *read-only* and *append-only*, to protect executable files and system activity logs which are helpful for intrusion detection.

In SVSDS, we leverage the idea of Type-Safe Disks (TSD) [16] to obtain higher-level semantic knowledge at the disk-level with minimal modifications to storage software such as file systems. By instrumenting file systems to automatically communicate logical block pointers to the disk system, a TSD can obtain three key pieces of information that are vital for implementing flexible security policies. First, by identifying blocks that have outgoing pointers, a TSD differentiates between data and meta-data. Second, a TSD differentiates between used and unused blocks, by just identifying blocks that have no incoming pointers (and hence not reachable from any meta-data block). Third, a TSD knows higher abstractions such as files and directories by just enumerating blocks in a sub-tree of the pointer hierarchy. For example, the sub-tree of blocks starting from an inode block of an Ext2 file system belong to a collection of files.

Using this semantic knowledge, SVSDS aggressively versions all meta-data blocks, as meta-data impact the accessibility of normal data, and hence is more important. It also provides an interface through which administrators can choose specific files or directories for versioning, or for enforcing operation-based constraints (read-only or append-only). SVSDS uses its knowledge of free and used blocks to place older versions of meta-data and chosen data, and virtualizes the block address-space. Older versions of blocks are not accessible to higher layers, except through a secure administrative interface upon authentication using a capability.

We implemented a prototype of SVSDS in the Linux kernel as a pseudo-device driver and evaluated its correctness and performance. Our results show that the overheads of selective disk-level versioning is quite minimal. For a normal user workload SVSDS had a small overhead of 1% compared to regular disks.

The rest of the paper is organized as follows. Section 2 describe background. Section 3 discusses the threat model. Section 4 and Section 5 explain the design and implementation of our system respectively. In Section 6, we discuss the performance evaluation of our prototype implementation. Related work is discussed in Section 7 and we conclude in Section 8.

## 2 Background

Data protection has been a major focus of systems research in the past decade. Inadvertent user errors, malicious intruders, and malware applications that exploit vulnerabilities in operating systems have exacerbated the need for stronger data protection mechanisms. In this section we first talk about versioning as a means for protecting data. We then give a brief description about TSDs to make the paper self-contained.

### 2.1 Data Versioning

Versioning data is a widely accepted solution to data protection especially for data recovery. Versioning has been implemented in different layers. It has been implemented above the operating system (in applications), inside the operating system (e.g., in file systems) and beneath the operating system (e.g., inside the disk firmware). We now discuss the advantages and disadvantages of versioning at the different layers.

**Application-level versioning.** Application-level versioning is primarily used for source code management [1, 2, 22]. The main advantage of these systems is that they provide the maximum flexibility as users can control everything from choosing the versioning application to creating new versions of files. The disadvantage with these systems is that they lack transparency and users can easily bypass the versioning mechanism. The versioned data is typically stored in a remote server and becomes vulnerable when the remote server's OS gets compromised.

**File-system–level versioning.** Several file systems support versioning [6, 10, 12, 15, 19]. These systems are mainly designed to allows users to access and revert back to previous versions of files. The older versions of files are typically stored under a hidden directory beneath its parent directory or on a separate partition. As these file systems maintain older versions of files, they can also be used for recovering individual files and directories in the event of an intrusion. Unlike application-level versioning systems, file-system–level versioning is usually transparent to higher layers. The main advantage of these versioning systems is that they can selectively version files and directories and can also support flexible versioning policies (e.g., users can choose different policies for each file or directory). Once a file is marked for versioning by the user, the file system automatically starts versioning the file data. The main problem with file-system–level versioning is that their security is closely tied to the security of the operating system. When the operating system is compromised, an intruder can bypass the security checks and change the data stored in the disk.

**Disk-level versioning.** The other alternative is to version blocks inside the disk [7, 20, 23]. The main advantage of this approach is that the versioning mechanism is totally decoupled from the operating system and hence can make data recoverable even when the operating system is compromised. The disadvantage with block-based disk-level versioning systems is that they cannot selectively version files as they lack semantic information about the data stored inside them. As a result, in most cases they end up versioning all the data inside the disk which causes them to have significant amount of space overheads in storing versions.

In summary, application-level versioning is weak in terms of security as can be easily bypassed by users. Also, the versioning mechanism is not transparent to users and can be easily disabled by intruders. File-system—level data-protection mechanisms provide transparency and also flexibility in terms of what data needs to be versioned but they do not protect the data in the event of an operating system compromise. Disk-level versioning systems provide better security than both application and file system level versioning but they do not provide any flexibility to the users to select the data that needs to be versioned. What we propose is a **hybrid solution**, i.e., combine the strong security that the disk-level data versioning provide, with the flexibility of file-system—level versioning systems.

## 2.2 Type-Safe Disks

Today's block-based disks cannot differentiate between block types due to the limited expressiveness of the block interface. All higher-level operations such as file creation, deletion, extension, renaming, etc. are translated into a set of block read and write requests. Hence, they do not convey any semantic knowledge about the blocks they modify. This problem is popularly known as the information gap in the storage stack [4, 5], and constrains disk systems with respect to the range of functionality that they can provide.

Pointers are the primary mechanisms by which data is organized. Most importantly, pointers define reachability of blocks; i.e., a block that is not pointed to by any other block cannot be reached or accessed. Almost all popular data structures used for storing information use pointers. For example, file systems and database systems make extensive use of pointers to organize the data stored in the disk. Storage mechanisms employed by databases like indexes, hash, lists, and b-trees use pointers to convey relationships between blocks.

Pointers are the smallest unit through which file systems organize data into semantically meaningful entities such as files and directories. Pointers define three things: (1) the semantic dependency between blocks; (2) the logical grouping of blocks; and (3) the importance of blocks. Even though pointers provide vast amounts of information about relationships among blocks, today's disks are oblivious to pointers. A Type-Safe Disk (TSD) is a disk system that is aware of pointer information and can use it to enforce invariants on data access and also perform various semantic-aware optimizations which are not possible in today's disk systems.

TSDs widen the traditional block-based interface to enable the software layers to communicate pointer information to the disk. File systems that use TSDs should use the disk APIs (CREATE_PTR, DELETE_PTR, ALLOC_BLOCK, GETFREE) exported by TSDs to allocate blocks, create and delete pointers, and get free-space information from the disk.

The pointer manager in TSDs keeps track of the relationship among blocks stored inside the disk. The pointer operations supported by TSDs are CREATE_PTR and DELETE_PTR. Both operations take two arguments: source and destination block numbers. The pointer manager uses a P-TABLE (or pointer table) to maintain the relationship among blocks inside the disk. Entries are added to and deleted from the P-TABLE during CREATE_PTR and DELETE_PTR operations. When there are no incoming pointers to a block it is automatically garbage collected by the TSD.

One other important difference between a regular disk and a TSD is that the file systems no longer does free-space management (i.e., file systems no longer need to maintain bitmaps to manage free space). The free-space management is entirely moved to the disk. TSDs export ALLOC_BLOCK API to allow file systems to request new blocks from the disk. The ALLOC_BLOCK API takes a reference block number, a hint block number, and the number of blocks as arguments and allocates the requested number of file system blocks from the disk maintained free block list. After allocating the new blocks, TSD creates pointers from the reference block to each of the newly allocated blocks.

The garbage-collection process performed in TSDs is different from the traditional garbage-collection mechanism employed in most programming languages. A TSD reclaims back the deleted blocks in an online fashion as opposed to the traditional offline mechanism in most programming languages. TSDs maintain a reference count (or the number of incoming pointers) for each block. When the reference count of a block decreases to zero, the block is garbage-collected; the space is reclaimed by the disk and the block is added to the list of free blocks. It is important to note that it is the pointer information provided by TSD that allows the disk to track the liveness of blocks, which cannot be done in traditional disks [17].

## 3 Threat Model

Broadly, SVSDS provides a security boundary at the disk level and makes vital data recoverable even when an attacker obtains root privileges. In our threat model, applications and the OS are untrusted, and the storage subsystem comprising the firmware and magnetic media is trusted. The OS communicates with the disk through a narrow interface that does not expose the disk internal versioning data. Our model assumes that the disk system is physically secure, and the disk protects against attackers that compromise a computer system through the network. This scenario covers a major class of attacks inflicted on computer systems today.

Specifically, an SVSDS provides the following guarantees:

- All meta-data and chosen file data marked for protection will be recoverable to an arbitrary previous state even if an attacker maliciously deletes or overwrites the data, after compromising the OS. The depth of history available for recovery is solely dependent on the amount of free-space available on disk. Given the fact that disk space is cheap, this is an acceptable dependency.

- Data items explicitly marked as *read-only* is guaranteed to be intact against any malicious deletion or overwriting.
- Data items marked as *append-only* can never be deleted or overwritten by any OS attacker.

It is important to note that SVSDS is designed to protect the data stored on the disk and does not provide any guarantee on which binaries/files are actually executed by the OS (e.g., rootkits could change the binaries in memory). As files with operation-based constraints (specifically read-only constraints) cannot be modified inside SVSDS, upon a reboot, the system running on SVSDS would return to a safe state (provided the system executables and configuration files are marked as read-only).

## 4 Design

Our aim while designing SVSDS is to combine the security of disk-level versioning, with the flexibility of versioning at higher-layers such as the file system. By transparently versioning data at the disk-level, we make data recoverable even in the event of OS compromises. However, today's disks lack information about higher-level abstractions of data (such as files and directories), and hence cannot support flexible versioning granularities. To solve this problem, we leverage Type-Safe Disks (TSDs) [16] and exploit higher-level data semantics at the disk-level.

Type-safe disks export an extended block-based interface to file systems. In addition to the regular block `read` and `write` primitives exported by traditional disks, TSDs support pointer management primitives that can be used by file systems to communicate pointer-relationships between disk blocks. For example, an Ext2 file system can communicate the relationships between an inode block of a file and its corresponding data blocks. Through this, logical abstractions of most file systems can be encoded and communicated to the disk system. Figure 1 shows the on-disk layout of Ext2. As seen from Figure 1, files and directories can be identified using pointers by just enumerating blocks of sub-trees with inode or directory blocks as root.

The overall goals of SVSDS are the following:

- Perform block versioning at the disk-level in a completely transparent manner such that higher-level software (such as file systems or user applications) cannot bypass it. System administrators or users can set up versioning policies or revert and delete versions through an offline privileged channel after a capability-based authentication process enforced by the disk system.

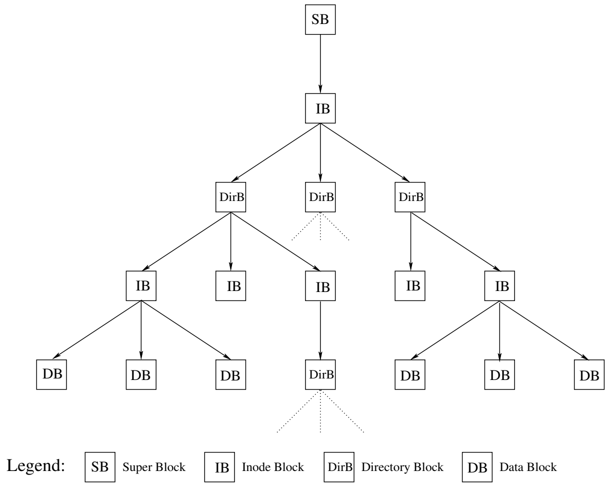


*Figure 1: Pointer relationship inside an FFS-like file system*

- Aggressively version all meta-data (e.g., Ext2 inode blocks) and chosen data as per the policies set up by administrators or users. In the perspective of a file system, versioning policies must be at granularities of individual files or directories.
- Enforce basic constraints at the disk-level, such as *read-only* and *append-only*. Users must be able to choose specific files or directories to be protected by these constraints.

Figure 2 shows the overall architecture of SVSDS. The three major components in SVSDS are, (1) Storage virtualization Layer (SVL), (2) The Version Manager, and (3) The Constraint Manager. The SVL virtualizes the block address space and manages physical space on the device. The version manager automatically versions meta-data and user-selected files and directories. It also provides an interface to revert back the disk state to previous versions. The constraint manager enforces read-only and append-only operation-level constraints on files and directories inside the disk.

The rest of this section is organized as follows. Section 4.1 describe how transparent versioning is performed inside SVSDS. Section 4.2 talks about the versioning mechanism. Section 4.4 describes our recovery mechanism and how an administrator recovers after detecting an OS intrusion. Section 4.5 describes how SVSDS enforces operation based constraints on files and

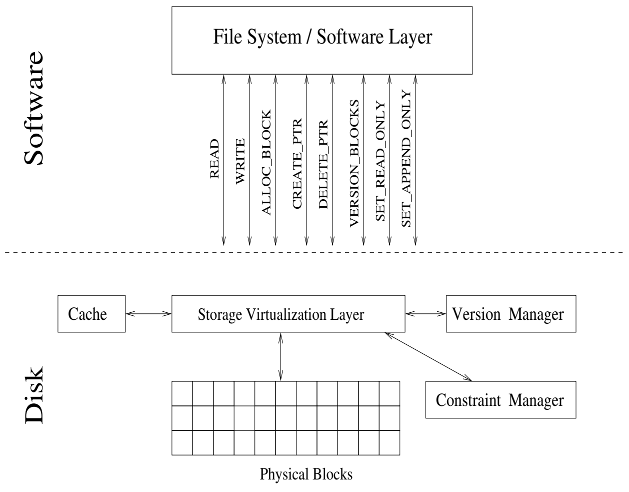


*Figure 2: Architecture of SVSDS*

directories. Finally, in Section 4.6, we discuss some of the issues with SVSDS.

## 4.1 Transparent Versioning

Transparent versioning is an important requirement, as SVSDS has to ensure that the versioning mechanism is not bypassed by higher layers. To provide transparent versioning, the storage virtualization layer (SVL) virtualizes the disk address space. The SVL splits the disk address space into two: logical and physical, and internally maintains the mapping between them. The logical address space is exposed to file systems and the SVL translates logical addresses to physical ones for every disk request. This enables SVL to transparently change the underlying physical block mappings when required, and applications are completely oblivious to the exact physical location of a logical block.

SVSDS maintains T-TABLE (or translation table), to store the relationship between logical and physical blocks. There is a one-to-one relationship between each logical and physical block in the T-TABLE. A version number field is also added to each entry of T-TABLE to denote the last version in which a particular block was modified. Also, a status flag is added to each T-TABLE entry to indicate the type (meta-data or data), and status (versioned or non-versioned) of each block. The T-TABLE is indexed by the logical block number and every allocated block has an entry in the T-TABLE. When applications read (or write) blocks, the SVL looks up the T-TABLE for the logical block and redirects the request to the corresponding physical block stored in the T-TABLE entry.

**Free-Space Management** SVSDS has two different address spaces, whereas the regular TSDs only have one. Hence, SVSDS cannot reuse the existing block allocation mechanism of regular TSDs. To manage both address spaces, the SVL uses two different bitmaps: logical block bitmaps (LBITMAPS) in addition to the existing physical block bitmaps (PBITMAPS). SVSDS uses a two-phased block allocation process. During the first phase, the SVL allocates the requested number of physical blocks from PBITMAPS. The allocation request need not always succeed as some of the physical blocks are used for storing the previous versions of blocks. If the physical block allocation request succeeds, it proceeds to the next phase. In the second phase, the SVL allocates an equal number of logical blocks from LBITMAPS. It then associates each of the newly allocated logical block with a physical block and adds an entry in the T-TABLE for each pair. The flags for these new entries are copied from the reference block passed to the ALLOC_BLOCK call and the version number is copied from the disk maintained version number. This ensures that all blocks that are added later to a file inherit the same attributes (or flags) as their parent block.

## 4.2 Creating versions

The version manager is responsible for creating new versions and maintaining previous versions of data on the disk. The version manager provides the flexibility of file-system–level versioning while operating inside the disk. By default, it versions all meta-data blocks. In addition, it can also selectively version user-selected files and directories. The version manager automatically checkpoints the meta-data and chosen data blocks at regular intervals of time, and performs copy-on-write upon subsequent modifications to the data. The version manager maintains a global version number and increments it after every checkpoint interval. The checkpoint interval is the time interval after which the version number is automatically incremented by the disk. SVSDS allows an administrator to specify the checkpoint interval through its administrative interface.

The version manager maintains a table, V-TABLE (or version table), to keep track of previous versions of blocks. For each version, the V-TABLE has a separate list of logical-to-physical block mappings for modified blocks.

Once the current version is checkpointed, any subsequent write to a versioned block creates a new version for that block. During this write, the version manager also backs up the existing logical to physical mapping in the V-TABLE. To create a new version of a block, the version manger allocates a new physical block through the SVL, changes the corresponding logical block entry in the T-

TABLE to point to the newly allocated physical block, and updates the version number of this entry to the current version. Figure 3 shows a V-TABLE with a few entries in the mapping list for the first three versions. Let's take a simple example to show how entries are added to the V-TABLE. If block 3 is overwritten in version 2, the entry in the T-TABLE for block 3 is added to the mapping list of the previous version (i.e., version 1).

**Versioning TSD Pointer Structures** TSDs maintains their own pointer structures inside the disk to track block relationships. The pointer management in TSDs was explained in Section 2.2. The pointers refers to the disk-level pointers inside TSDs, unless otherwise mentioned in the paper. As pointers are used to track block liveness information inside TSDs, the disk needs to keep its pointer structures up to date at all times. When the disk is reverted back to the previous version, the pointer operations performed in the current version have to be undone for the disk to reclaim back the space used by the current version.

To undo the pointer operations, SVSDS logs all pointer operations to the pointer operation list of the current version in the V-TABLE. For example, in Figure 3 the first entry in the pointer operation list for version 1 shows that a pointer was created between logical blocks 3 and 8. This create pointer operation has to be undone when the disk is reverted back from version 1 to 0. Similarly, the first entry in the pointer operation list for version 3 denotes that a pointer was deleted between logical blocks 3 and 8. This operation has to be undone when the disk is reverted back from version 3 to version 2.

To reduce the space required to store the pointer operations, SVSDS does not store pointer operations on blocks created and deleted (or deleted and created) within the same version. When a CREATE_PTR is issued with source $a$ and destination $b$ in version $x$. During the lifetime of the version $x$, if a DELETE_PTR operation is called with the same source $a$ and destination $b$, then the version manager removes the entry from the pointer operation list for that version in the V-TABLE. We can safely remove these pointer operations because CREATE_PTR and DELETE_PTR operations are the inverse of each other and would cancel out their changes when they occur withing the same version. The recovery manager maintains a hash table indexed on the source and destination pair for efficient retrieval of entries from the V-TABLE.

## 4.3 Selective Versioning

Current block-based disk systems lack semantic information about the data being stored inside. As a result, disk-level versioning systems [7, 23] version all blocks. But versioning all blocks inside the disk can quickly consume all available free space on the disk. Also, versioning all blocks is not efficient for the following two reasons: (1) short lived temporary data (e.g., data in the */tmp* folder and installation programs) need not be versioned, and (2) persistent data blocks have varying levels of importance. For example, in FFS-like file systems, versioning the super block, inode blocks, or indirect blocks is more important than versioning data blocks as the former affects the reachability of other blocks stored inside the disk. Hence, SVSDS selectively versions meta-data and user-selected files and directories to provide deeper version histories.

**Versioning meta-data.** Meta-data blocks have to be versioned inside the disk for two reasons. First, reachability: meta-data blocks affects the reachability of data blocks that it points to (e.g., the data blocks can only be reached through the inode or the indirect block). Second, recovery of user-selected files: we need to preserve all versions of the entire file system directory-structure inside the disk to revert back files and directories.

To selectively version meta-data blocks, SVSDS uses the pointer information available inside the TSDs. SVSDS identifies a meta-data block during the first CREATE_PTR operation the block passed as the source is identified as a meta-data block. For all source block passed to the CREATE_PTR operation, SVSDS marks it as meta-data in the T-TABLE.

SVSDS defers reallocation of deleted data blocks until there are no free blocks available inside the disk. This ensures that for a period of time the deleted data blocks will still be valid and can be restored back when their corresponding meta-data blocks are reverted back during recovery.

To version files and directories, applications issue an ioctl to the file system that uses SVSDS. The file system in turn locates the logical block number of the file's inode block, and calls the VERSION_BLOCKS disk primitive. VERSION_BLOCKS is a new primitive added to the existing disk interface for applications to communicate the files for versioning (see Table 1). After the blocks of the file are marked for versioning, the disk automatically versions the marked blocks at regular intervals.

**Versioning user-selected data.** Versioning meta-data blocks alone does not make the disk system more secure. Users still want the disk to automatically version certain files and directories. To selectively version files and directories, applications and file systems only have to pass the starting block (or the root of the subtree) under which all the blocks needs to be versioned. For example, in Ext2 only the inode block of the file or the directory needs to be passed for versioning. SVSDS does

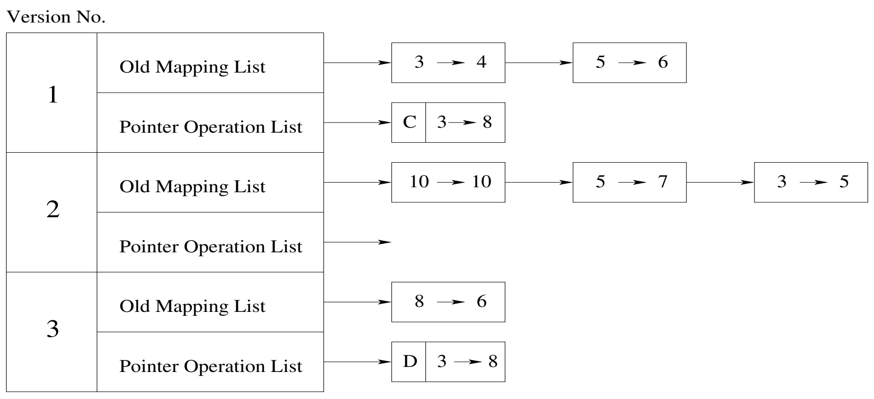


*Figure 3:* **The v-table data structure.** *A simplified v-table state is shown for first three versions in SVSDS. Each entry in the old mapping list corresponds to logical and physical block pair. C & D in the pointer operation list represent Create pointer and Delete pointer operations, respectively.*

a Breadth First Search (BFS) on the P-TABLE, starting from the root of the subtree. All the blocks traversed during the BFS are marked for versioning in the T-TABLE.

One common issue in performing BFS is that there could potentially be many cycles in the graph that is being traversed. For example, in the Ext2TSD [16] file system, there is a pointer from the inode of the directory block, to the inode of the sub-directory block and vice versa. Symbolic links are yet another source of cycles. SVSDS detects cycles by maintaining a hash table (D-TABLE) for blocks that have been visited during the BFS. During each stage of the BFS, the version manager checks to see if the currently visited node is present in the D-TABLE before traversing the blocks pointed to by this block. If the block is already present in the D-TABLE, SVSDS skips the block as it was already marked for versioning. If not, SVSDS adds the currently visited block to the D-TABLE before continuing with the BFS.

To identify blocks that are subsequently added to versioned files or directories, SVSDS checks the flags present in the T-TABLE of the source block during the CREATE_PTR operations. This is because when file systems want to get a free block from SVSDS, they issue an ALLOC_BLOCK call with a reference block and the number of required blocks as arguments. This ALLOC_BLOCK call is internally translated to a CREATE_PTR operation with the reference block and the newly allocated block as its arguments. If the reference block is marked to be versioned, then the destination block that it points to is also marked for versioning. File systems normally pass the inode or the indirect block as the reference block.

## 4.4 Reverting Versions

In the event of an intrusion or an operating system compromise, an administrator would want to undo the changes done by an intruder or a malicious application by reverting back to a previous safe state of the disk. We define reverting back to a previous versions as restoring the disk state from time $t$ to the disk state at time $t$ - $tv$, where $tv$ is the checkpoint interval.

Even though SVSDS can access any previous version's data, we require reverting only one version at a time. This is because SVSDS internally maintains state about block relationships through pointers, and it requires that the pointer information be properly updated inside the disk to garbage-collect deleted blocks. To illustrate the problem with reverting back to an arbitrary version, let's revert the disk state from version $f$ to version $a$ by skipping reverting of the versions between $f$ and $a$. Reverting back the V-TABLE entries for version $a$ alone would not suffice. As we directly jump to version $a$, the blocks that were allocated, and pointers that were created or deleted between versions $f$ and $a$, are not reverted back. The blocks present during version $a$ does not contain information about blocks created after version $a$. As a result, blocks allocated after version $a$ becomes unreachable by applications but according to pointer information in the P-TABLE they are still reachable. As a result, the disk will not reclaim back these block and the we will be leaking disk space. Hence, SVSDS allows an administrator to revert back only one version at a time.

SVSDS also allows an administrator to revert back the disk state to a arbitrary point in time by reverting back one version at a time until the largest version whose start time is less than or equal to the time mentioned by the administrator is found. REVERT_TO_PREVIOUS_VERSION and REVERT_TO_TIME

| **Disk Primitives** | **Description** |
|---|---|
| VERSION_BLOCKS$(BNo)$ | Marks all blocks in the subtree starting from block $BNo$ to be versioned. The data blocks present in the subtree will be versioned along with the reference (or meta-data) blocks. |
| REVERT_TO_PREVIOUS_VERSION | Reverts back the disk state from current version to the previous version. |
| REVERT_TO_TIME$(t)$ | Reverts back the disk state one version at a time till it finds a version $v$ with start time less than or equal to $t$. |
| MARK_READ_ONLY$(BNo)$ | Marks all blocks in the sub-tree starting from block $BNo$ as read-only. |
| MARK_APPEND_ONLY$(BNo)$ | Marks all blocks in the sub-tree starting from block $BNo$ as append-only. $BNo$ itself will not be an append-only block as it could be a meta-data block, with non-sequential updates. |

Table 1: *Additional Disk APIs in SVSDS*

are the additional primitives added to the existing disk interface to revert back versions by the administrator (see Table 1).

While reverting back to a previous version, SVSDS recovers the data by reverting back the following: (1) *Pointers*: the pointer operation that happened in the current version are reverted back; (2) *Meta-data*: all meta-data changes that happened in the current version are reverted back; (3) *Data-blocks*: all versioned data blocks and some (or all) of the non-versioned deleted data-blocks are reverted back (i.e., the non-versioned data blocks that have been garbage collected cannot be reverted back); and (4) *Bitmaps*: both logical and physical block bitmap changes that happened during the current version are reverted.

### 4.4.1 Reverting Mapping

SVSDS reverts back to its previous version from the current version in two phases. In the first phase, it restores all the T-TABLE entries stored in the mapping list of the previous version in the V-TABLE. While restoring back the T-TABLE entries of the previous version, there are two cases that need to be handled. (1) An entry already exists in the T-TABLE for the logical block of the restored mapping. (2) An entry does not exist. When an entry exists in the T-TABLE, the current mapping is replaced with the old physical block from the mapping list in the V-TABLE. The current physical block is freed by clearing the bit corresponding to the physical block number in the PBITMAPS. If an entry does not exist in the T-TABLE, it implies that the block was deleted in the current version and the mapping was backed up in the V-TABLE. SVSDS restores the mapping as a new entry in the T-TABLE and the logical block is marked as used in the LBITMAPS. The physical block need not be marked as used as it is already alive. At the end of the first phase, SVSDS restores back all the versioned data that got modified or deleted in the current version.

### 4.4.2 Reverting Pointer Operations

In the second phase of the recovery process, SVSDS reverts back the pointer operations performed in the current version by applying the inverse of the pointer operations. The inverse of the CREATE_PTR operation is a DELETE_PTR operation and vice versa. The pointer operations are reverted back to free up the space used by blocks created in the current version and also for restoring pointers deleted in the current version.

Reverting back CREATE_PTR operations are straight forward. SVSDS issues the corresponding DELETE_PTR operations. If there are no incoming pointers to the destination blocks of the DELETE_PTR operations, the disk automatically garbage collects the destination blocks.

While reverting the DELETE_PTR operations, SVSDS checks if the destination blocks are present in the T-TABLE. If yes, SVSDS executes the corresponding CREATE_PTR operations. If the destination blocks is not present in the T-TABLE, it implies that the DELETE_PTR operations were performed on non-versioned blocks. If the destination blocks are present in the deleted block list, SVSDS restores the backed up T-TABLE entries from the deleted block list and issues the corresponding CREATE_PTR operations.

While reverting back to a previous version, the inverse pointer operations have to be replayed in the reverse order. If not, SVSDS would prematurely garbage collect these blocks. We illustrate this problem with a simple example. From Figure 4(a) we can see that block $a$ has a pointer to block $b$ and block $b$ has pointers to blocks $c$ and $d$. The pointers from $b$ are first deleted and then the pointer from $a$ to $b$ is deleted. This is shown in Figs. 4(b) and 4(c). If the inverse pointer operations are applied in the same order, first a pointer would be is created from block $b$ to $d$ (assuming pointer from $b$ to $d$ is deleted first) but block $b$ would be automatically garbage collected by SVSDS as there are no incoming pointers to block $b$. Replaying pointer operations in the reverse order avoids this problem. Figs 4(d), 4(e), and 4(f) show the sequence of

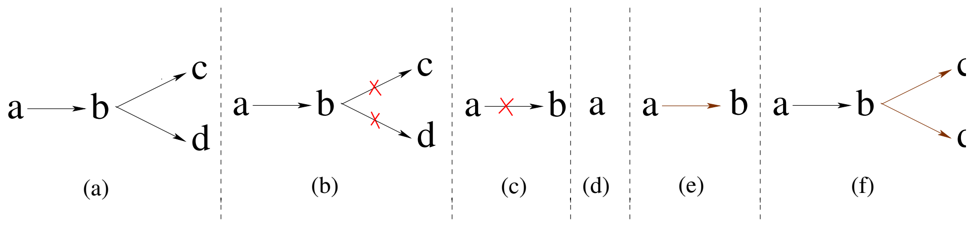


*Figure 4:* **Steps in reverting back delete pointer operations**

steps performed while reverting back the delete pointer operations in the reverse order. We can see that reverting back pointer operations in the reverse order correctly reestablishes the pointers in the correct sequence.

### 4.4.3 Reverting Meta-Data

SVSDS uses the mapping information in the V-TABLE to revert back changes to the meta-data blocks. There are three cases that need to be handled while reverting back meta-data blocks: (1) The meta-data block is modified in the new version, (2) The meta-data block is deleted in the new version, and (3) The meta-data block is first modified and then deleted in the new version. In the first case, the mappings that are backed up in the previous version for the modified block in the V-TABLE are restored. This is done to get back the previous contents of the meta-data blocks. For the second case, the delete pointer operations would have caused the T-TABLE entries to be backed up in the V-TABLE as they would be the last incoming pointer to the meta-data blocks. The T-TABLE entries will be restored back in the first phase of the recovery process and the deleted pointers are restored back in the second phase of the recovery process. Reverting meta-data blocks when they are first modified and then deleted is the same as in reverting meta-data blocks when they are deleted.

### 4.4.4 Reverting Data Blocks

When the recovery manager reverts back to a previous version, it cannot revert back to the exact disk state in most cases. To revert back to the exact disk state, the disk would need to revert mappings for all blocks, including the data blocks that are not versioned by default. In a typical TSD scenario, blocks are automatically garbage collected as soon as the last incoming pointer to them is deleted, making their recovery difficult if not impossible. The garbage collector in SVSDS tries to reclaim the deleted data blocks as late as possible. To do this, SVSDS maintains an LRU list of deleted non-versioned blocks (also known as the deleted block list).

When the delete-pointer operations are reverted back, SVSDS issues the corresponding create-pointer operations only if the deleted data blocks are still present in the deleted block list. This policy of lazy garbage collection allows users to recover the deleted data blocks that have not yet been garbage collected yet.

Lazy garbage collection is also useful when a user reverts back the disk state after inadvertently deleting a directory. If all data blocks that belong to the directory are not garbage collected, then the user can get back the entire directory along with the files stored under it. If some of the blocks are already reclaimed by the disk, the user would get back the deleted directory with data missing in some files. Even though SVSDS does not version all data block, it still tries to restore back all deleted data blocks when disk is revert back to its previous version.

### 4.4.5 Reverting Bitmaps

When data blocks are added or reclaimed back during the recovery process the bitmaps have to be adjusted to keep track of free blocks. The PBITMAPS need not be restored back as they are never deleted. The physical blocks are backed up either in the deleted block list or in the old mapping lists in the V-TABLE. The physical blocks that are added in the current version are freed during the first and second phases of the recovery process. During the first phase, the previous version's data is restored from mapping list in the V-TABLE. At this time the physical blocks of the newer version are marked free in the PBITMAPS. When the pointers created in the current version are reverted back by deleting them in the second phase, the garbage collector frees both the physical and the logical blocks, only if it is the last incoming pointer to the destination block.

The LBITMAPS only have to be restored back for versioned blocks that have been deleted in the current version. While restoring the backed up mappings from the V-TABLE, SVSDS checks if the logical block is allocated in the LBITMAPS. If it is not allocated, SVSDS reallocates the deleted logical block by setting the corresponding bit in the LBITMAPS. The deleted non-versioned blocks need not be restored back. Previously, these blocks were moved to the deleted block list and were added back to the T-TABLE during the second phase of the recovery process.

## 4.5 Operation-based constraints

In addition to versioning data inside the disk, it is also important to protect certain blocks from being modified, overwritten, or deleted. SVSDS allows users to specify the types of operations that can be performed on a block, and the constraint manager enforces these constraints during block writes. SVSDS enforces two types of operation-based constraints: read-only and append-only.

The sequence of steps taken by the operation manager to mark a file as read-only or append-only is the same as marking a file to be versioned. The steps for marking a file to be versioned was described in Section 4.3. While marking a group of blocks, the first block (or the root block of the subtree) encountered in the breadth first search is treated differently to accommodate special file system updates. For example, file systems under UNIX support three timestamps: access time (atime), modification time (mtime), and creation time (ctime). When data from a file is read, its atime is updated in the file's inode. Similarly, when the file is modified, its mtime and ctime are updated in its inode. To accommodate atime, mtime, and ctime updates on the first block, the constraint manager distinguishes the first block by adding a special meta-data block flag in the T-TABLE for the block. SVSDS disallows deletion of blocks marked as read-only or append-only constraints. MARK_READ_ONLY and MARK_APPEND_ONLY are the two new APIs that have been added to the disk for applications to specify the operation-based constraints on blocks stored inside the disk. These APIs are described in Table 1.

**Read-only constraint.** The read-only operation-based constraint is implemented to make block(s) immutable. For example, the system administrator could mark binaries or directories that contain libraries as read-only, so that later on they are not modified by an intruder or any other malware application. Since SVSDS does not have information about the file system data structures, atime updates cannot be distinguished from regular block writes using pointer information. SVSDS neglects (or disallows) the atime updates on read-only blocks, as they do not change the integrity of the file. Note that the read-only constraint can also be applied to files that are rarely updated (such as binaries). When such files have to be updated, the read-only constraint can be removed and set back again by the administrator through the secure disk interface.

**Append-only constraint.** Log files serve as an important resource for intrusion analysis and statistics collection. The results of the intrusion analysis is heavily dependent on the integrity of the log files. The operation-based constraints implemented by SVSDS can be used to protect log files from being overwritten or deleted by intruders.

SVSDS allows marking any subtree in the pointer chain as "append-only". During a write to a block in an append-only subtree, the operation manager allows it only if the modification is to change trailing zeroes to non-zeroes values. SVSDS checks the difference between the original and the new contents to verify that data is only being appended, and not overwritten. To improve the performance, the operation manager caches the append-only blocks when they are written to the disk to avoid reading the original contents of block from the disk during comparison. If a block is not present in the cache, the constraint manager reads the block and adds it to the cache before processing the write request. To speed up comparisons, the operation manager also stores the offsets of end of data inside the append-only blocks. The newly written data is compared with the cached data until the stored offsets.

When data is appended to the log file, the atime and the mtime are also updated in the inode block of the file by the file system. As a result, the first block of the append-only block is overwritten with every update to the file. As mentioned earlier, SVSDS does not have the information about the file system data structures. Hence, SVSDS permits the first block of the append-only files to be overwritten by the file system.

SVSDS does not have information about how file systems organize its directory data. Hence, enforcing append-only constraints on directories will only work iff the new directory entries are added after the existing entries. This also ensures that files in directories marked as append-only cannot be deleted. This would help in preventing malicious users from deleting a file and creating a symlink to a new file (for example, an attacker can no longer unlink a critical file like */etc/passwd*, and then just creates a new file in its place).

## 4.6 Issues

In this section, we talk about some of the issues with SVSDS. First we talk about the file system consistency after reverting back to a previous version inside the disk. We then talk about the need for a special port on the disk to provide secure communication. Finally, we talk about Denial of Service (DoS) attacks and possible solutions to overcome them.

**Consistency** Although TSDs understand a limited amount of file system semantics through pointers, they are still oblivious to the exact format of file system-specific meta-data and hence it cannot revert the state that is consistent in the viewpoint of specific file systems. A file system consistency checker (e.g., *fsck*) needs to be run after the disk is reverted back to a previous version. Since SVSDS internally uses pointers to track blocks, the consistency checker should also issue appropriate calls to SVSDS to ensure that disk-level pointers are consistent with file system pointers.

**Administrative Interfaces** To prevent unauthorized users from reverting versions inside the disk, SVSDS should have a special hardware interface through which an administrator can log in and revert back versions. This port can also be used for setting the checkpoint frequency.

**Supporting Encryption File Systems** Encryption File systems (EFS) can run on top of SVSDS with minimal modifications. SVSDS only requires EFS to use TSD's API for block allocation and notifying pointer relationship to the disk. The append-only operation-based constraint would not work for EFS as end of block cannot be detected if blocks are encrypted. If encryption keys are changed across versions and if the administrator reverts back to a previous version, the decryption of the file would no longer work. One possible solution is to change the encryption keys of files after a capability based authentication upon which SVSDS would decrypt all the older versions and re-encrypt them with the newly provided keys. The disadvantage with this approach is that the versioned blocks need to be decrypted and re-encrypted when the keys are changed.

**DoS Attacks** SVSDS is vulnerable to denial of service attacks. There are three issues to be handled: (1) blocks that are marked for versioning could be repeatedly overwritten; (2) lots of bogus files could be created to delete old versions, and (3) versioned files could be deleted and recreated again preventing subsequent modifications to files from being versioned inside the disk. To counter attacks of type 1, SVSDS can throttle writes to files that are versioned very frequently. An alternative solution to this problem would be to exponentially increase the versioning interval of the particular file / directory that is being constantly overwritten resulting in fewer number of versions for the file. As with most of the denial of service attacks there is no perfect solution to attack of type 2. One possible solution would be to stop further writes to the disk, until some of the space used up by older versions, are freed up by the administrator through the administrative interface. The downside of this approach is that the disk effectively becomes read-only till the administrator frees up some space. Type 3 attacks are not that serious as versioned files are always backed up when they are deleted. One possible solution to prevent versioned files from being deleted is to add *no-delete* flag on the inode block of the file. This flag would be checked by SVSDS along with other operation-based constraints before deleting/modifying the block. The downside of this approach is that normal users can no longer delete versioned files that have been marked as *no-delete*. The administrator has to explicitly delete this flag on the *no-delete* files.

## 5 Implementation

We implemented a prototype SVSDS as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. Figure 5 shows the pseudo device driver implementation of SVSDS. SVSDS has 7,487 lines of kernel code out of which 3,060 were reused from an existing TSD prototype. The SVSDS layer receives all block requests from the file system, and re-maps and redirects the common read and write requests to the lower-level device driver. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctl`s.

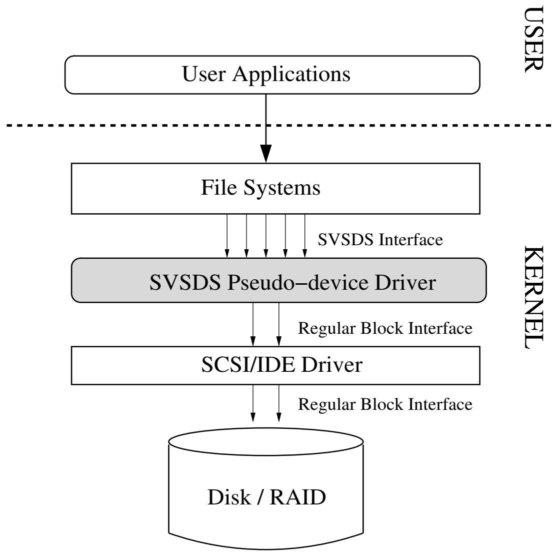


Figure 5: Prototype Implementation of SVSDS

In the current implementation we maintain all hash tables (V-TABLE, T-TABLE, P-TABLE, and D-TABLE) as in-memory data structures. As these hash tables only have small space requirements, they can be persistently stored in a portion of the NVRAM inside the disk. This helps SVSDS to avoid disk I/O for reading these tables.

The read and write requests from file systems reach SVSDS through the Block IO (BIO) layer in the Linux kernel. The BIO layer issues I/O requests with the destination block number, callback function (BI_END_IO), and the buffers for data transfer, embedded inside the BIO data structure. To redirect the block requests from SVSDS to the underlying disk, we add a new data structure (BACKUP_BIO_DATA). This structure stores the destination block number, BI_END_IO, and BI_PRIVATE of the BIO data structure. The BI_PRIVATE field is used by the owner of the BIO request to store private information. As I/O request are by default asynchronous in the Linux kernel, we stored the original contents of the BIO data structures by replacing the value stored inside BI_PRIVATE to point to our BACKUP_BIO_DATA data structure. When I/O requests reach SVSDS, we replace the destination block number, BI_END_IO, and BI_PRIVATE in the BIO data structure with the mapped physical block from the T-TABLE, our callback function (SVSDS_END_IO), and the BACKUP_BIO_DATA respectively. Once the I/O request is completed, the control reaches our SVSDS_END_IO function. In this function, we restore back the original block number and BI_PRIVATE information from the BACKUP_BIO_DATA data structure. We then call the BI_END_IO function stored in the BACKUP_BIO_DATA data structure, to notify the BIO layer that the I/O request is now complete.

We did not make any design changes to the existing Ext2TSD file system to support SVSDS. The Ext2TSD is a modified version of the Ext2 file system that notifies the pointer relationship to the file system through the TSD disk APIs. To enable users to select files and directories for versioning or enforcing operation-based constraints, we have added three ioctls namely: VERSION_FILE, MARK_FILE_READONLY, and MARK_FILE_APPENDONLY to the Ext2TSD file system. All three ioctls take a file descriptor as their argument, and gets the inode number from the in-memory inode data structure. Once the Ext2TSD file system has the inode number of the file, it finds the the logical block number that correspond to inode number of the file. Finally, we call the the corresponding disk primitive from the file system ioctl with logical block number of the inode as the argument. Inside the disk primitive we mark the file's blocks for versioning or enforcing operation-based constraint by performing a breadth first search on the P-TABLE.

## 6 Evaluation

We evaluated the performance of our prototype SVSDS using the Ext2TSD file system [16]. We ran general-purpose workloads on our prototype and compared them with unmodified Ext2 file system on a regular disk. This section is organized as follows: In Section 6.1, we talk about our test platform, configurations, and procedures. Section 6.2 analyzes the performance of the SVSDS framework for an I/O-intensive workload, Postmark [8]. In Sections 6.3 and 6.4 we analyze the performance on OpenSSH and kernel compile workloads respectively.

### 6.1 Test infrastructure

We conducted all tests on a 2.8GHz Intel Xeon CPU with 1GB RAM, and a 74GB 10Krpm Ultra-320 SCSI disk. We used Fedora Core 6 running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-$t$ distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the difference between elapsed time and CPU time, and is affected by I/O and process scheduling.

Unless otherwise mentioned, the system time overheads were mainly caused by the hash table lookups on T-TABLE during the read and write operations and also due to P-TABLE lookups during CREATE_PTR and DELETE_PTR operations. This CPU overhead is due to the fact that our prototype is implemented as a pseudo-device driver that runs on the same CPU as the file system. In a real SVSDS setting, the hash table lookups will be performed by the processor embedded in the disk and hence will not influence the overheads on the host system, but will add to the wait time.

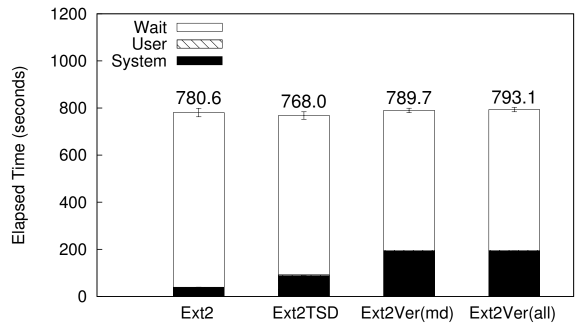
We have compared the overheads of SVSDS using Ext2TSD against Ext2 on a regular disk. We denote Ext2TSD on a SVSDS using the name Ext2Ver. The letters $md$ and $all$ are used to denote selective versioning of meta-data and all data respectively.

### 6.2 Postmark

Postmark [8] simulates the operation of electronic mail and news servers. It does so by performing a series of file system operations such as appends, file reads, directory lookups, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 3,000 files, between 100–200 kilobytes, and perform 300,000 transactions.

Figure 6 show the performance of Ex2TSD on SVSDS for Postmark with a versioning interval of 30 seconds. Postmark deletes all its files at the end of the benchmark, so no space is occupied at the end of the test. SVSDS transparently creates versions and thus, consumes storage space which is not visible to the file system. The average number of versions created during this benchmark is 27.

For Ext2TSD, system time is observed to be 1.1 times more, and wait time is 8% lesser that of Ext2. The



| | Ext2 | Ext2TSD | Ext2Ver(md) | Ext2Ver(all) |
|---|---|---|---|---|
| Elapsed | 780.5s | 768.0s | 789.7s | 793.1s |
| System | 36.28s | 88.58s | 191.71s | 191.94s |
| Wait | 741.42s | 676.11s | 593.80s | 597.09s |
| Space o/h | 0MB | 0MB | 443MB | 1879MB |
| Performance Overhead over Ext2 | | | | |
| Elapsed | - | -1.60 % | 1.17% | 1.61% |
| System | - | 1.44 × | 4.28 × | 4.29× |
| Wait | - | -8.12 % | -19.91% | -19.47% |

Figure 6: Postmark results for SVSDS

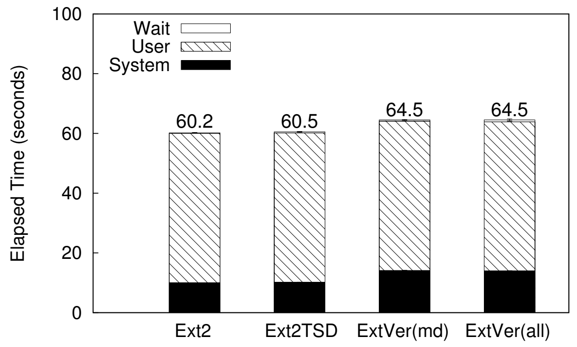
increase in the system time is because of the hash table lookups during CREATE_PTR and DELETE_PTR calls. The decrease in the wait time is because, Ext2TSD does not take into account future growth of files while allocating space for files. This decrease in wait time allowed Ext2TSD to perform slight better than Ext2 file system on a regular disk, but would have had a more significant impact in a benchmark with files that grow.

For Ext2Ver(md), elapsed time is observed to have no overhead, system time is 4 times more and wait time is 20% less than that of Ext2. The increase in system time is due to the additional hash table lookups to locate entries in the T-TABLE. The decrease in wait time is due to better spacial locality and increased number of requests being merged inside the disk. This is because the random writes (i.e., writing inode block along with writing the newly allocated block) were converted to sequential writes due to copy-on-write in versioning.

For Ext2Ver(all), The system time is 4 times more and wait time is 20% less that of Ext2. The wait time in Ext2Ver(all) does not have any observable overhead over the wait time in Ext2Ver(md). Hence, it is not possible to explain for the slight increase in the wait time.

## 6.3 OpenSSH Compile

To show the space overheads of a typical program installer, we compiled the OpenSSH source code. We used OpenSSH version 4.5, and analyzed the overheads of Ext2 on a regular disk, Ext2TSD on a TSD, and meta-data and all data versioning in Ext2TSD on SVSDS for the `untar`, `configure`, and `make` stages combined. Since the entire benchmark completed in 60–65 seconds, we used a 2 second versioning interval to create more versions of blocks. On an average, 10 versions were created. This is because the pdflush deamon starts writing the modified file system blocks to disk after 30 seconds. As a result, the disk does not get any write request for blocks during the first 30 seconds of the OpenSSH Compile benchmark. The amount of data generated by this benchmark was 16MB. The results for the OpenSSH compilation are shown in Figure 7.



| | Ext2 | Ext2TSD | Ext2Ver(md) | Ext2Ver(all) |
|---|---|---|---|---|
| Elapsed | 60.186s | 60.532s | 64.520s | 64.546s |
| System | 10.027s | 10.231s | 14.147s | 14.025s |
| Wait | 0.187s | 0.390s | 0.454s | 0.634s |
| Space o/h | 0MB | 0MB | 496KB | 15.14MB |
| Performance Overhead over Ext2 | | | | |
| Elapsed | - | 0.57 % | 7.20% | 7.21% |
| System | - | 2 % | 41 % | 39% |
| Wait | - | 108 % | 142% | 238% |

Figure 7: OpenSSH Compile Results for SVSDS

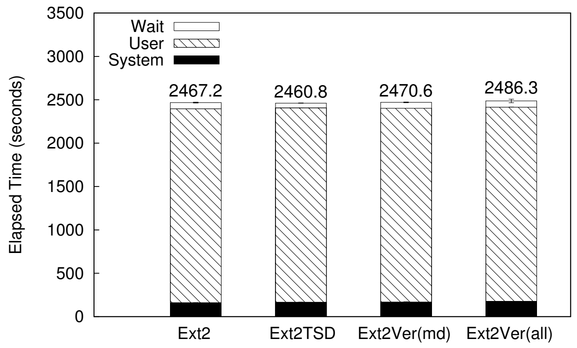
For Ext2TSD, we recorded a insignificant increase in elapsed time and system time, and a 108% increase in the wait time over Ext2. Since the elapsed and system times are similar, it is not possible to quantify for the increase in wait time.

For Ext2Ver(md), we recorded a 7% increase in elapsed time, and a 41% increase in system time over Ext2. The increase in system time overhead is due to the additional hash table lookups by SVL to remap the read and write requests. Ext2Ver(md) consumed 496KB of additional disk space to store the versions.

For Ext2Ver(all), we recorded a 7% increase in elapsed time, and a 39% increase in system time over Ext2. Ext2Ver(all) consumes 15MB of additional space to store the versions. The overhead of storing versions is 95%. From this benchmark, we can clearly see that the versioning all data inside the disk is not very useful, especially for program installers.

## 6.4 Kernel Compile

To simulate a CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel and analyzed the overheads of Ext2TSD on a TSD and Ext2TSD on SVSDS with versioning of all blocks and selective versioning of meta-data blocks against regular Ext2, for the `untar`, `make oldconfig`, and `make` operations combined. We used 30 second versioning interval and 78 versions were created during this benchmark. The results are shown in Figure 8.



| | Ext2 | Ext2TSD | Ext2Ver(md) | Ext2Ver(all) |
|---|---|---|---|---|
| Elapsed | 2467s | 2461s | 2471s | 2468s |
| System | 162s | 167s | 169s | 177s |
| Wait | 72.1s | 54.7s | 68.0s | 71.6s |
| Space o/h | 0MB | 0MB | 51MB | 181MB |
| Performance Overhead over Ext2 | | | | |
| Elapsed | - | -0.26 % | 0.13% | 0.77% |
| System | - | 3.6% | 4.7% | 10% |
| Wait | - | -24% | -5.6% | -0.8% |

Figure 8: Kernel Compile results for SVSDS.

For Ext2TSD, elapsed time is observed to be the same, system time overhead is 4% lower and wait time is lower by 24% than that of Ext2. The decrease in the wait time is because Ext2TSD does not consider future growth of files while allocating new blocks.

For Ext2Ver(md), elapsed time is observed to be the same, system time overhead is 5%, and wait time is lower by 6% than that of Ext2. The increase in wait time in relation to ext2TSD is due to versioning meta-data blocks which affect the locality of the stored files. The space overhead of versioning meta-data blocks is 51 MB.

For Ext2Ver(all), elapsed time is observed to be indistinguishable, system time overhead is 10% higher than that of Ext2. The increase in system time is due to the additional hash table lookups required for storing the mapping information in the V-TABLE. The space overhead of versioning all blocks is 181 MB.

## 7 Related Work

SVSDS borrows ideas from many of the previous works. The idea of versioning at the granularity of files has been explored in many file systems [6, 10, 12, 15, 19]. These file systems maintain previous versions of files primarily to help users to recover from their mistakes. The main advantage of SVSDS over these systems is that, it is decoupled from the client operating system. This helps in protecting the versioned data, even in the event of an intrusion or an operating system compromise. The virtualization of disk address space has been implemented in several systems [3, 7, 9, 13, 21]. For example, the Logical disk [3] separated the file-system implementation from the disk characteristics by providing a logical view of the block device. The Storage Virtualization Layer in SVSDS is analogous to their logical disk layer. The operation-based constraints in SVSDS is a scaled down version of access control mechanisms. We now compare and contrast SVSDS with other disk-level data protection systems: S4 [20], TRAP [23], and Peabody [7].

The Self-Securing Storage System (S4) is an object-based disk that internally audits all requests that arrive at the disk. It protects data in compromised systems by combining log-structuring with journal-based meta-data versioning to prevent intruders from tampering or permanently deleting the data stored on the disk. SVSDS on the other hand, is a block-based disk that protect data by transparently versioning blocks inside the disk. The guarantees provided by S4 hold true only during the window of time in which it versions the data. When the disk runs out of storage space, S4 stops versioning data until the cleaner thread can free up space for versioning to continue. As S4 is designed to aid in intrusion diagnosis and recovery, it does not provide any flexibility to users to version files (i.e, objects) inside the disk. In contrast, SVSDS allows users to select files and directories for versioning inside the disk. The disadvantage with S4 is that, it does not provide any protection mechanism to prevent modifications to stored data during intrusions and always depends on the versioned data to recover from intrusions. In contrast, SVSDS attempts to prevent modifications to stored data during intrusions by enforcing operation-based constraints on system and log files.

Timely Recovery to any Point-in-time (TRAP) is a disk array architecture that provides data recovery in three different modes. The three modes are: TRAP-1 that takes snapshots at periodic time intervals; TRAP-3 that provides timely recovery to any point in time at the block device level (this mode is popularly known as Continuous Data Protection in storage); TRAP-4 is similar to RAID-5, where a log of the parities is kept for each block write. The disadvantage with this system is that, it cannot provide TRAP-2 (data protection at the file-level) as their block-based disk lacks semantic information about the data stored in the disk blocks. Hence, TRAP ends up versioning all the blocks. TRAP-1 is similar to our current implementation where an administrator can choose a particular interval to version blocks. We have implemented TRAP-2, or file-level versioning inside the disk as SVSDS has semantic information about blocks stored on the disk through pointers. TRAP-3 is similar to the mode in SVSDS where the time between creating versions is set to zero. Since SVSDS runs on a local disk, it cannot implement the TRAP-4 level of versioning.

Peabody is a network block storage device, that virtualizes the disk space to provide the illusion of a single large disk to the clients. It maintains a centralized repository of sectors and tries to reduce the space utilization by coalescing blocks across multiple virtual disks that contain the same data. This is done to improve the cache utilization and to reduce the total amount of storage space. Peabody versions data by maintaining write logs and transaction logs. The write logs stores the previous contents of blocks before they are overwritten, and the transaction logs contain information about when the block was written, location of the block, and the content hashes of the blocks. The disadvantage with this approach is that it cannot selectively versions blocks inside the disk.

## 8 Conclusions

Data protection against attackers with OS root privileges is fundamentally a hard problem. While there are numerous security mechanisms that can protect data under various threat scenarios, only very few of them can be effective when the OS is compromised. In view of the fact that it is virtually impossible to eliminate all vulnerabilities in the OS, it is useful to explore how best we can recover from damages once a vulnerability exploit has been detected. In this paper, we have taken this direction and explored how a disk-level recovery mechanism can be implemented, while still allowing flexible policies in tune with the higher-level abstractions of data. We have also shown how the disk system can enforce simple constraints that can effectively protect key executables and log files. Our solution that combines the advantages of a software and a hardware-level mechanism proves to be an effective choice against alternative methods. Our evaluation of our prototype implementation of SVSDS shows that performance overheads are negligible for normal user workloads.

**Future Work** . Our current design supports reverting the entire disk state to an older version. In future, we plan to work on supporting more fine-grained recovery policies to revert specific files or directories to their older versions. SVSDS in its current form, relies on the administrator to detect an intrusion and revert back to a previously known safe state. We plan to build a storage-based intrusion detection system [14] inside SVSDS. Our system would do better than the system developed by Pennington et al. [14] as we also have data dependencies conveyed through pointers. We also plan to explore more operation-based constraints that can be supported at the disk-level.

## 9 Acknowledgments

We like to thank the anonymous reviewers for their helpful comments. We thank Sean Callanan and Avishay Traeger for their feedback about the project. We would also like to thank the following people for their comments and suggestions on the work: Radu Sion, Rob Johnson, Radu Grosu, Alexander Mohr, and the members of our research group (File systems and Storage Lab at Stony Brook).

This work was partially made possible by NSF CAREER EIA-0133589 and NSF CCR-0310493 awards.

## References

[1] B. Berliner and J. Polk. Concurrent Versions System (CVS). `www.cvshome.org`, 2001.

[2] CollabNet, Inc. Subversion. `http://subversion.tigris.org`, 2004.

[3] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM SIGOPS.

[4] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[5] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, CMU, December 2001.

[6] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[7] C. B. Morrey III and D. Grunwald. Peabody: The time travelling disk. In *Proceedings of the 20 th*

*IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 241–253. IEEE Computer Society, 2003.

[8] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 84–92, Cambridge, MA, 1996.

[10] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[11] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41, August 2003. ieeexplore.ieee.org.

[12] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[13] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, June 1988.

[14] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, pages 137–152, Washington, DC, August 2003.

[15] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 2–7, Rio Rica, AZ, March 1999.

[16] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[17] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[18] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[19] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, March 2003. USENIX Association.

[20] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 165–180, San Diego, CA, October 2000. USENIX Association.

[21] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 185–197, Boston, MA, June 2001. USENIX Association.

[22] Walter F. Tichy. RCS — a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[23] Q. Yang, W. Xiao, and J. Ren. TRAP-array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, pages 289–301. IEEE Computer Society, 2006.

# Privacy-Preserving Location Tracking of Lost or Stolen Devices: Cryptographic Techniques and Replacing Trusted Third Parties with DHTs

*Thomas Ristenpart*[*] *Gabriel Maganis*[†] *Arvind Krishnamurthy*[†] *Tadayoshi Kohno*[†]

[*]*University of California, San Diego*
`tristenp@cs.ucsd.edu`

[†]*University of Washington*
`{gym,arvind,yoshi}@cs.washington.edu`

## Abstract

We tackle the problem of building *privacy-preserving device-tracking systems* — or private methods to assist in the recovery of lost or stolen Internet-connected mobile devices. The main goals of such systems are seemingly contradictory: to hide the device's legitimately-visited locations from third-party services and other parties (*location privacy*) while simultaneously using those same services to help recover the device's location(s) after it goes missing (*device-tracking*). We propose a system, named Adeona, that nevertheless meets both goals. It provides strong guarantees of location privacy while preserving the ability to efficiently track missing devices. We build a version of Adeona that uses OpenDHT as the third party service, resulting in an immediately deployable system that does not rely on any single trusted third party. We describe numerous extensions for the basic design that increase Adeona's suitability for particular deployment environments.

## 1 Introduction

The growing ubiquity of mobile computing devices, and our reliance upon them, means that losing them is simultaneously more likely and more damaging. For example, the annual CSI/FBI Computer Crime and Security Survey ranks laptop and mobile device theft as a prevalent and expensive problem for corporations [16]. To help combat this growing problem, corporations and individuals are deploying commercial *device-tracking* software — like "LoJack for Laptops" [1] — on their mobile devices. These systems typically send the identity of the device and its current network location (e.g., its IP address) over the Internet to a central server run by the device-tracking service. After losing a device, the service can determine the location of the device and, subsequently, can work with the owner and legal authorities to recover the device itself. The number of companies offering such services, e.g., [1, 9, 21, 29, 34, 37, 38], attests to the large and growing market for device tracking.

Unfortunately, these systems are incompatible with the oft-cited goal of *location privacy* [17, 22, 23] since the device-tracking services can always monitor the location of an Internet-enabled device — even while the device is in its owner's possession. This presents a significant barrier to the psychological acceptability of tracking services. To paraphrase one industry representative: companies will deploy these systems in order to track their devices, but they won't like it. The current situation leaves users of mobile devices in the awkward position of *either* using tracking services *or* protecting their location privacy.

We offer an alternative: *privacy-preserving device-tracking systems*. Such a system should provide strong guarantees of location privacy for the device owner's legitimately visited locations while nevertheless enabling tracking of the device after it goes missing. It should do so even while relying on untrusted third party services to store tracking updates.

**The utility of device tracking systems.** Before diving into technical details, we first step back to reevaluate whether device tracking, let alone privacy-preserving device tracking, even makes sense as a legitimate security tool for mobile device users. A motivated and sufficiently equipped or knowledgeable thief (i.e., the malicious entity assumed in possession of a missing device) can *always* prevent Internet device tracking: he or she can erase software on the device, deny Internet access, or even destroy the device. One might even be tempted to conclude that the products of [1, 9, 21, 29, 34, 37, 38] are just security "snake oil".

We purport that this extreme view of security is inappropriate for device tracking. While device tracking will not always work, these systems *can* work, and vendors (who may be admittedly biased) claim high recov-

[*]Work done while at the University of Washington.

ery rates [1]. The common-case thief is, after all, often opportunistic and unsophisticated, and it is against such thieves that tracking systems can clearly add significant value. Our work aims to retain this value while simultaneously addressing the considerable threats to user location privacy.

**System goals.** A device tracking system consists of: client hardware or software logic installed on the device; (sometimes) cryptographic key material stored on the device; (sometimes) cryptographic key material maintained separately by the device owner; and a remote storage facility. The client sends *location updates* over the Internet to the remote storage. Once a device goes missing, the owner or authorized agent searches the remote storage for location updates pertaining to the device's current whereabouts.

To understand the goals of a privacy-preserving tracking system, we begin with an exploration of existing or hypothetical tracking systems in scenarios that are derived from real situations (Section 2). This reveals a restrictive set of deployment constraints (e.g., supporting both efficient hardware and software clients) and an intricate threat model for location privacy where the remote storage provider is untrusted, the thief may try to learn past locations of the device, and other outsiders might attempt to glean private data from the system or "piggyback" on it to easily track a device. We extract the following main system goals.

(1) Updates sent by the client must be *anonymous* and *unlinkable*. This means that no adversary should be able to either associate an update to a particular device, or even associate two updates to the same (unknown) device.

(2) The tracking client must ensure *forward-privacy*, meaning a thief, even after seeing all of the internal state of the client, cannot learn past locations of the device.

(3) The client should protect against *timing attacks* by ensuring that the periodicity of updates cannot be easily used to identify a device.

(4) The owner should be able to efficiently search the remote storage in a privacy-preserving manner.

(5) The system must match closely the efficiency, deployability, and functionality of existing solutions that have little or no privacy guarantees.

These goals are not satisfied by straightforward or existing solutions. For example, simply encrypting location updates before sending to the remote storage does not allow for efficient retrieval. As another example, mechanisms for generating secure audit logs [32], while seemingly applicable, in fact violate our anonymity and unlinkability requirements by design.

We emphasize that one non-goal of our system is *improved* device tracking. As discussed above, all tracking systems in this category have fundamental limitations. Indeed, our overarching goal is to show that, in any setting where deploying a device tracking system makes sense, one can do so effectively *without compromising privacy*.

**Adeona.** Our system, named Adeona after the Roman goddess of "safe returns," meets the aggressive goals outlined above. The client consists of two modules: a location-finding module and a cryptographic core. With a small amount of state, the core utilizes a forward-secure pseudorandom generator (FSPRG) to efficiently and deterministically encapsulate updates, rendering them anonymous and unlinkable, while also scheduling them to be sent to the remote storage at pseudorandomly determined times (to help mitigate timing attacks). The core ensures forward-privacy: a thief, after determining all of the internal state of the client and even with access to all data on the remote storage, cannot use Adeona to reveal past locations of the device. The owner, with a copy of the initial state of the client, can efficiently search the remote storage for the updates. The cryptographic core uses only a sparing number of calls to AES per update.

The cryptographic techniques in the Adeona core have wide applicability, straightforwardly composing with any location-finding technique or remote storage instantiation. We showcase this by implementing Adeona as a fully functional tracking system using a public distributed storage infrastructure, OpenDHT [30]. We could also have potentially used other distributed hash table infrastructures such as the Azureus BitTorrent DHT. Using a DHT for remote storage means that there is no single trusted infrastructural component and that deployment can proceed immediately in a community-based way. End users need simply install a software client to enable private tracking service. Our system provides the first device tracking system not tied to a particular service provider. Moreover, to the best of our knowledge, we are also the first to explore replacing a centralized trusted third-party service with a decentralized DHT.

**Extensions.** Adeona does make slight trade-offs between simplicity, privacy, and device tracking. We address these trade-offs with several extensions to the basic Adeona system. These extensions serve two purposes: they highlight the versatility of our basic privacy-enhancing techniques and they can be used to better protect the tracking client against technically sophisticated thieves (at the cost of slight increases in complexity). In particular, we discuss several additions to the basic functionality of Adeona. For example, we design a novel cryptographic primitive, a tamper-evident FSPRG, to allow detection of adversarial modifications to the client's state.

**Implementation and field testing.** We have implemented the Adeona system and some of its extensions as user applications for Linux and Mac OS X. Moreover, we conducted a short trial in which the system was deployed on real users' systems, including a number of laptops. Our experience suggests that the Adeona system provides an immediate solution for privacy-preserving device tracking. The code is currently being readied for an open-source public release to be available at `http://adeona.cs.washington.edu/`, and we encourage the further use of this system for research purposes.

**Outline.** In the next section we provide a detailed discussion of tracking scenarios that help motivate our (involved) design constraints and threat models. Readers eager for technical details might skip ahead to Section 3, which describes the Adeona core. The full system based on OpenDHT is given in Section 4. We provide a security analysis in Section 5. Our implementations, their evaluation, and the results of the field trial appear in Section 6. We discuss Adeona's suitability for further deployment settings in Section 7 and extensions to Adeona are detailed in Section 8. We conclude in Section 9.

## 2 Problem Formulation

To explore existing and potential tracking system designs and understand the variety of adversarial threats, we first study a sequence of hypothetical tracking scenarios. While fictional, the scenarios are based on real stories and products. These scenarios uncover issues that will affect our goals and designs for private device tracking.

**Scenario 1.** Vance, an avid consumer of mobile devices, recently heard about the idea of "LoJack for Laptops." He searches the Internet, finds the EmailMe device tracking system, and installs it on his laptop.[1] The EmailMe tracking client software sends an email (like the example shown in Figure 1) to his webmail account every time the laptop connects to the Internet. Months later, Vance is distracted while working at his favorite coffee shop, and a thief takes his laptop. Now Vance's foresight appears to pay off: he uses a friend's computer to access the tracking emails sent by his missing laptop. Working with the authorities, they are able to determine that the laptop last connected to the Internet from a public wireless access point in his home city. Unfortunately the physical location was hard to pinpoint from just the IP addresses. A month after the theft Vance stops receiving tracking emails. An investigation eventually reveals that the thief sold the laptop at a flea market to an unsuspecting customer.[2] That customer later resold the laptop at a pawn shop. The pawnbroker, before further reselling the laptop, must have refurbished the laptop by wiping its hard drive and installing a fresh version of the operating system.

**Discussion:** The theft of Vance's laptop highlights a few issues regarding limitations on the functionality of device tracking systems. First, a client without hardware-support can provide network location data only when faced by such a *flea-market attack*: these occur when a technically unsophisticated thief steals a device to use it or sell it (with its software intact) as quickly as possible. Second, network location information will not always be sufficient for precisely determining the *physical location* of a device. Third, all clients (even those with hardware support) can be disabled from sending location updates (simply by disallowing all Internet access or by filtering out just the location updates if they can be isolated).

The principal goal of this paper is not to achieve better Internet tracking functionality than can be offered by existing solutions. Instead, we address privacy concerns while maintaining device tracking functionality equivalent to solutions with no or limited privacy guarantees. The next scenarios highlight the types of privacy concerns inherent to tracking systems.

**Scenario 2.** A few weeks before the theft of Vance's laptop, Vance was the target of a different kind of attack. His favorite coffee shop had been targeted by crackers because the shop is in a rich neighborhood and their routers are not configured to use WPA [19]. The crackers recorded all the coffee shop's traffic, including Vance's location-update emails, which were not encrypted. (The webmail service did not use TLS, nor does the EmailMe client encrypt the outgoing emails.) The crackers sell the data garnered from Vance's tracking emails to identity thieves, who then use Vance's identity to obtain several credit cards.

**Discussion:** The content of location updates should always be sent via *encrypted channels*, lest they reveal private information to passive eavesdroppers. This is of particular importance for mobile computing devices, because of their almost universal use of wireless communication, which may or may not use encryption.

**Scenario 3.** Vance works as a salesman for a small distributor of coffee-related products, called Very Good Coffee (VGC). He recently went on a trip abroad for VGC to investigate purchasing a supplier of coffee beans. On his return trip, he was stopped at customs and his laptop was temporarily confiscated for an "inspection" [28, 33]. Vance, with his ever-present foresight, had predicted this would happen: he encrypted all his sensitive work-related files and removed any information that might leak what he had been doing while in country. The laptop was shortly returned with files apparently unmodified.

Unknown to Vance, the EmailMe client had cached

```
From:  tech@brigadoonsoftware.com                          Mac Address:  00-18-8B-A2-05-E5
To:  tech@brigadoonsoftware.com                            Mac Address:  00-18-DE-9B-F0-5A
BCC: tomrist@gmail.com                                     Serial Number:  DC44BF26
Subject:  Information                                      Registrants Name:  Tom
PCPH Pro For Win 95/98/ME/NT/2K/XP - Version 3.0 (Eval)    Organization:  Tom
                                                           Address:  513 Brooklyn Avenue
Date:  16-08-2007                                          City:  Seattle
Time:  11:14:05                                            State/Province:  WA
Computer Name :  TOM-8F760D01401                           Zip/Postal Code:  98105
User Name :  LOCAL SERVICE                                 Country:  USA
IPAddress :0.0.0.0                                         Work Phone:  2066163997
IPAddress :128.208.7.80                                    ...
```

Figure 1: Example tracking email sent (unencrypted) by PC Phone Home [9] from one of the authors' laptops.

all the recently visited network locations on the laptop. Included were several IP addresses used by the supplier that VGC intended to purchase. The customs agents sold this information to a local competitor of VGC. Using this tip, the local competitor successfully blocked VGC's bid to purchase the supplier.

**Discussion:** This scenario addresses the need for *forward privacy*. A tracking client should not cache previous locations, lest a thief (or even, as the scenario depicts, some other untrusted party with temporary access to the device) easily break the owner's past location privacy.

**Scenario 4.** Hearing about Vance's recent troubles with property and identity theft, the VGC management chose to contract with (the optimistically named) All Devices Recovered (AllDevRec) to provide robust tracking services for VGC's mobile assets. AllDevRec, having made deals with laptop manufacturers, ensures that VGC's new laptops have hardware-supported tracking clients installed. The clients send updates using a proprietary protocol over an an encrypted channel to AllDevRec's servers each time an Internet connection is made.[3]

Ian, a recovery-management technician employed by AllDevRec, has a good friend Eve who happens to work at a business that competes with VGC. Ian brags to Eve that his position in AllDevRec allows him to access the locations from which VGC's employees access the Internet. This gives Eve an idea, and so she goads Ian into giving her information on the network locations visited by VGC sales people. From this Eve can infer the coffee shops VGC is targeting as potential customers, allowing her company to precisely undercut VGC's offerings.

**Discussion:** Using encrypted channels is insufficient to guarantee data privacy once the location updates reach a service provider's storage systems. The location updates should remain *encrypted while stored*. This mitigates the level of trust device owners must place in a service provider's ability to enforce proper data management policies (to protect against insider attacks) and security mechanisms (to protect against outsiders gaining access).

**Scenario 5.** Vance, now jobless due to VGC's recent bankruptcy, has been staying at Valerie's place. Valerie works at a large company, with its own in-house IT staff. The management decided to deploy a comprehensive tracking system for mobile computing asset management. To ensure employee acceptability of a tracking system, the management had the IT staff implement a system with privacy and security issues in mind: each device is assigned a random identification number and a public key, secret key pair for a public-key encryption scheme. The database mapping a device to its identification number, public key, and secret key is stored on a system with several procedural safeguards in place to ensure no unwarranted accesses. With each new Internet connection, the tracking client sends an update encrypted under the public key and indexed under the random identification number.

When Valerie goes to lunch (which varies in time quite a bit depending on her work), she heads across the street to a cafe to get away from the office. She often uses her company laptop and the cafe's wireless to peruse the Internet. Since deployment of the new tracking system, Valerie has been complaining that no matter when she takes lunch, Irving (a member of the IT staff who is reputed to have an unreciprocated romantic interest in her) almost always ends up coming by the cafe a few minutes after she arrives.[4]

Because the location updates sent by Valerie's laptop use a static identifier, it was easy for Irving (even without access to the protected database) to infer which was hers: he looked at identifiers with updates originating from the block of IP addresses used within Valerie's department and those used by the cafe. After a few guesses (which he validated by simply seeing if she was at the cafe), Irving determined her device's identification number and from then on knew whenever she went for lunch.

**Discussion:** The use of unchanging identifiers (even if originally anonymized) allows *linking attacks*, in which an adversary observing updates can associate updates from different locations as being from the same device.

Additionally, the finely-grained timing information revealed by sending updates upon each new Internet connection is a side-channel that can leak information.

**Summary.** The sequence of scenarios depicts the wide variety of potential users of tracking systems. Moreover, they highlight two fundamental security goals.

- Vance was a victim of compromised *device tracking*. (Scenario 1.)
- Vance, VGC, and Valerie were all victims of compromised *location privacy*. (Scenarios 2, 3, 4, and 5.)

The threat models related to achieving location privacy while retaining device tracking capabilities are complex because there exist numerous adversaries with widely varied powers and motivation:

- The unscrupulous party in possession of a device, which we will simply call the *thief*. The thief might be unsophisticated, sophisticated and intent on disabling the tracking device, or sophisticated and wish to reveal past locations.
- Internet-connected *outsiders* that might intercept update traffic (e.g., the crackers at the coffee shop). Such adversaries call for ensuring the use of encrypted channels.
- The *remote storage provider*, or the entity controlling the system(s) that host location updates, might be untrustworthy, suggesting the need for location updates that are *anonymous, unlinkable, and encrypted*, thereby denying private information even to the remote storage provider.

## 3 The Adeona Core: Providing Anonymous, Unlinkable Updates

The core module is the portion of a client primarily responsible for preparing, scheduling, and sending location updates to the remote storage. The Adeona core is, consequently, the foundation of our tracking system's privacy properties. We treat its development first, and mention that the core stands by itself as a component that will work in numerous deployment settings, in addition to the setting handled by the full Adeona system (described in the next section).

The discussion in Section 2 illustrates that the Adeona core must provide mechanisms to

(1) ensure content sent to the remote storage is anonymous and unlinkable;

(2) ensure forward-privacy (stored data on the client should not be sufficient for revealing previous locations);

(3) mitigate timing attacks; and

(4) allow the owner to efficiently search the remote storage for updates.

**Basic design.** A first approach for building a core would be to just utilize a secure symmetric encryption scheme. That is, the owner could install on the client a secret key and also store a copy separately, perhaps printed on a piece of paper or stored on a secure removable token. For each new Internet connection, the core would encrypt the location data using this secret key and immediately send the ciphertext to the remote storage. Goal (1) above would be satisfied because (assuming one used a standard, secure encryption scheme) these ciphertexts would, indeed, be anonymous and unlinkable. But, the other three goals are *not* met. A thief that gets access to the device and the secret key could decrypt previous updates. Sending the ciphertext immediately upon detecting a new Internet connection also leaks fine-grained timing information. More importantly, since ciphertexts submitted by all users are anonymous, there is no efficient way for the owner to search the database for his updates.[5]

The Adeona core utilizes a more sophisticated approach to tackle the other goals while preserving the ability to address goal (1). Instead of a key for an encryption scheme, the owner initializes the client with a *secret cryptographic seed* for a pseudorandom generator (PRG) [6]. Each time the core is run it uses the PRG and the seed to deterministically generate two fresh pseudorandom values: an index and a secret key (for the encryption scheme). The location information is encrypted using the secret key. The core sends both the index and the ciphertext to the remote storage. As before the ciphertext reveals no information, but the index is pseudorandom as well, meaning the entire update is anonymous and unlinkable. Thus goal (1) is satisfied. Goal (4) is as well: the owner, having a copy of the original cryptographic seed, can recompute all of the indices and keys used. This allows for efficient search of the remote storage for his or her updates, using the indices. The indices do not reveal decryption keys nor past or future indices.

This approach does not yet satisfy goal (2), because a thief — or customs official — can also use the seed to generate all the past indices and keys. We can rectify this by using a *forward-secure pseudorandom generator* (FSPRG) [5]: instead of using a single cryptographic seed for the lifetime of the system, the core also evolves the seed pseudorandomly. When run, the core uses the FSPRG and the seed to generate an index, secret key, and a new seed. The old seed is discarded (securely erased). The properties of the FSPRG ensure that it is computationally intractable to "go backwards" so that previous seeds (and the associated indices and keys) remain unknown even to a thief with access to the current seed.

Finally we can address goal (3) by randomly select-

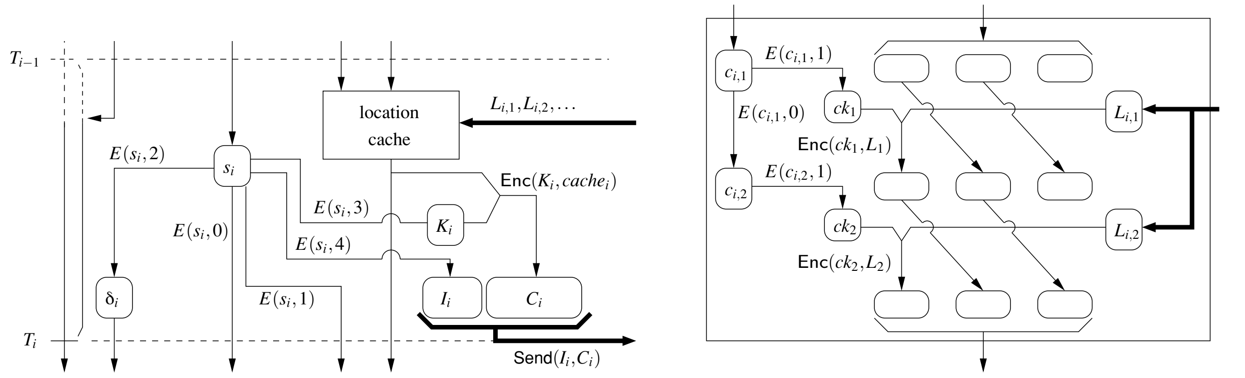


Figure 2: **(Left)** The Adeona core, where $E$ is a block cipher (e.g., AES) instantiating the FSPRG and Enc is a standard encryption scheme. **(Right)** Close-up of the core's forward-private location caching, where the cache holds 3 updates and shown are two new locations being stored.

ing times to send updates. Using the FSPRG as a source of randomness, we can pseudorandomly generate exponentially-distributed inter-update times. (This allows the owner to also recompute the inter-update times, which will be useful for retrieval as discussed in Section 4.) Such a distribution is memoryless, meaning that, from the storage provider's view, the next update is equally likely to come from any client. We can tune the number of updates sent by adjusting the rate of the exponential distribution used.

**Forward-private location caching.** Our pseudorandom update schedule means that we might miss locations that are visited for only a short amount of time. However, to provide maximal evidentiary forensic data about the trajectories of a device after theft, we would like the core to allow reporting all of the recently visited locations. We could cache recent locations, but this breaks forward-privacy. We therefore enhance the basic design to include a *forward-private location cache*. Having a cache also provides a simple mechanism for adding temporal redundancy to updates (i.e., location data is sent multiple times to the remote storage over time), which can increase the ability to successfully retrieve updates.

Instead of just caching location data in the clear, we can have the core immediately encrypt new data sent from the location-finding module. The resulting ciphertext can then be added to a cache; the least recent ciphertext is expelled. However, we cannot just utilize the encryption key generated by the current state's FSPRG: a thief could decrypt any ciphertexts in the cache that were added since the last time the FSPRG seed was refreshed (e.g., when the previous update was sent). We therefore use a distinct FSPRG seed, which we call the *cache seed*, as the source for generating encryption keys for each location encountered. Each time the cache seed is used to encrypt new location data, it is also used to generate a new cache seed and the prior one is securely erased. In this way we guarantee forward privacy: no data in the core allows a thief to decrypt previously generated ciphertexts. When its time to send an update, the entire cache is encrypted using the secret key generated by the FSPRG with the main seed. This (second) encryption ensures that the data stored at the remote storage cannot later be correlated with ciphertexts in the cache. Finally, the core "resets" the cache seed by generating a fresh one using the FSPRG and the main seed. This associates a sequences of cache seeds to a particular update state. We ensure freshness of location data by mandating that at least one newly generated ciphertext is included with each update submitted to the remote storage.

The owner can reconstruct all of the cache seeds for any state (using the prior state's main seed) and do trial decryption to recover locations. (The number of expected trials is the number of locations visited in between two updates, and so this will be typically small.) Ciphertexts in the cache that are "leftover" from a prior update time period can also be decrypted, and this can be rendered efficient if plaintexts include a hint (i.e., the number of states back) that specifies which state generated the keys for the next ciphertext entry.

**Implementing the design.** Implementing the Adeona core is straightforward, given a block cipher[6] such as AES. A standard and provably secure FSPRG implementation based on AES works as follows [5]. A cryptographic seed is just an AES key (16 bytes). To generate a string of pseudorandom bits, one iteratively applies AES, under key a seed $s$, to a counter: $\mathsf{AES}(s,1)$, $\mathsf{AES}(s,2)$, etc. For Adeona, we have an initial main seed $s_1$ and initial cache seed $c_{1,1}$ (both randomly generated). The main seed $s_1$ is used to generate a new seed $s_2 = \mathsf{AES}(s_1,0)$, the next state's cache seed $c_{2,1} = \mathsf{AES}(s_1,1)$, and so on for the encryption key, index, and time offset. (The exponentially distributed time offset is generated from a pseudorandom input using the well known method of inverse-transform sampling [13].) A seed, after it is used, must be securely erased. The cache seed forms a separate branch of the FSPRG and is used to generate a sequence of cache seeds and intermediate encryption keys for use within the cache. Figure 2 provides a diagram of the core module's operation between two successive updates at times $T_{i-1}$ and $T_i$.

The encryption scheme can also be built using just AES, via an efficient block cipher mode such as GCM [26]. Such a mode also provides authenticity. Of added benefit is that the mode can be rendered deterministic (i.e., no randomness needed) since we only encrypt a single message with each key. This means that the core (once initialized) does not require a source of true randomness.

**Summary.** To summarize, the core uses a sequence of secret seeds $s_1, s_2, \ldots$ to provide

- a sequence $I_1, I_2, \ldots$ of *pseudorandom indices* to store ciphertexts under,
- sequences $c_{i,1}, c_{i,2}, \ldots$ of *secret cache seeds* for each state $i$ that are then used to encrypt data about each location visited,
- a sequence $K_1, K_2, \ldots$ of *secret keys* for encrypting the cache before submission to the remote storage, and
- a sequence $\delta_1, \delta_2, \ldots$ of *pseudorandom inter-update times* for scheduling updates

while providing the following assurances. Given any $I_i$, $K_j$, or $\delta_l$, no adversary can (under reasonable assumptions) compute any of the other output values above. Additionally, even if the thief views the entire internal state of the core, it still cannot compute any of the core's previously used indices, cache seeds, encryption keys, or inter-update times.

## 4 The Adeona System: Private Tracking using OpenDHT

A (privacy-preserving) tracking system consists of three main components: the device, the remote storage; and an owner. The device component itself consists of a location-finding component and a core component; other components — such as a camera image capture functionality — can easily be incorporated. A system works in three phases: initialization, active use, and retrieval. We have already seen the Adeona core. In this section we show how to construct a complete privacy-preserving device tracking system using it.

Our target is to develop an open-source, immediately deployable system. This will allow evaluation of our techniques during real usage (see Section 6), not to mention providing to individual users an immediate (and, to our knowledge, first) alternative to the plethora of existing, proprietary tracking systems, none of which achieve the level of privacy that we target and that we believe will be important to many users. Along these lines, this section focuses on a model for a open source software-only client. We use the public distributed storage infrastructure OpenDHT [30] for the remote storage facility. Not only does this obviate the need to setup dedicated remote storage facilities, enabling immediate deployability, but it effectively removes our system's reliance on any single trusted third party. This adds significantly to the practical privacy guarantees of the system.

We now flesh out the design of the complete Adeona system. The client consists of the Adeona core of the previous section (with a few slight modifications described below) plus a location-finding module, described below. First, however, we describe the other components: using OpenDHT for remote storage and how to perform privacy-preserving retrieval. We conclude the section with a summary of the whole system.

**OpenDHT as remote storage.** A distributed hash table (DHT) allows insertion and retrieval of data values based on hash keys. OpenDHT is an implementation of a distributed hash table (DHT) whose nodes run on PlanetLab [11]. We use the indices generated by the Adeona core as the hash keys and store the ciphertext data under them. There are several benefits to using a public, open-source distributed hash table (DHT) as remote storage. First, existing DHT's such as OpenDHT are already deployed and usable, meaning deployment of the tracking system only requires distribution of software for the client and for retrieval. Second, a DHT can naturally provide strengthened privacy and security guarantees because of the fact that updates will be stored uniformly across all the nodes of the DHT. In decentralized DHTs, an attacker would have to corrupt a significant fraction of DHT nodes in order to mount Denial-of-Service or privacy attacks as the storage provider.

On the other hand, DHT's also have limitations. The most fundamental is a lack of persistence guarantee: the DHT itself provides no assurance that inserted data can always be retrieved. Fortunately, OpenDHT ensures that inserted data is retained for at least a week.[7] Another limitation is temporary connectivity problems. Often nodes, even in OpenDHT, can be difficult to access, meaning our client will not be able to send an update successfully. The traditional approaches for handling such issues is to use client-side replication. This means that the client submits the same data to multiple, widely distributed nodes in the DHT.

We can enhance the Adeona core to include such a replication mechanism easily: have the core generate several indices (as opposed to just one) for each update. These indices, being pseudorandom already, will be distributed uniformly across the the space of all DHT nodes.

The update can then be submitted under all of these indices.

**Scheduling location updates.** The Adeona core provides a method to search for update ciphertexts via the deterministically generated indices. As noted, querying the remote storage for a set of indices does not reveal decryption keys or past or future indices. However, just the fact that a set of indices are queried for might allow the remote storage provider to trivially associated them to the same device. While the distributed nature of OpenDHT mitigates this threat, defense-in-depth asks that we do better. We therefore want a mechanism that ensures the owner can precisely determine which indices to search for when performing queries, and in particular allow him to avoid querying indices used before the device was lost or stolen.

To enable this functionality, we have the system precisely (but still pseudorandomly) schedule updates relative to some clock. The clock could be provided, for example, by a remote time server that the client and owner can synchronize against. Then, when the owner initializes the client, in addition to picking the cryptographic seed it also stores the current time as the initial time stamp $T_1$. Each subsequent state also has a time stamp associated with it: $T_2$, $T_3$, etc. These indicate the state's scheduled send time, and $T_{i+1}$ is computed by adding $T_i$ and $\delta_i$ (the pseudorandom inter-update delay). When the client is run, it reads the current time from the clock and iterates past states whose scheduled send time have already past. (In this way the core will "catch up" the state to the schedule.) With access to a clock loosely synchronized against the client's, the owner can accurately retrieve updates sent at various times (e.g., last week's updates, all the updates after the device went missing, etc.). We discuss the assumption of a clock more in our security analysis in Section 5.

**Location-finding module.** Our system works modularly with any known location finding technique (e.g., determining external IP address, trace routes to nearby routers, GPS, nearby 802.11 or GSM beacons, etc.). We implemented three different location-finding mechanisms: light, medium, and full. The *light mechanism* just determines the internal IP address and the externally-facing IP address. (The latter being the IP as reported by an external server.) The *medium mechanism* additionally performs traceroutes to 8 randomly-chosen PlanetLab nodes. These traceroutes provide additional information about the device's current surrounding network topology. The *full mechanism* employs a protocol that adapts state-of-the-art geolocationing techniques to our setting. Here, geolocationing refers to determining (approximate) physical locations from network data. Traditional approaches utilize a distributed set of landmarks to actively probe a target [18]. These probes, combined with the knowledge of the physical locations of the landmarks, allows approximate geolocationing of the target. We flip this approach around, using the active-client nature of our setting to have the client itself find nearby *passive landmarks*.

Concretely, we utilize Akamai [2] nodes as landmarks: they are numerous, widespread, and often co-located within ISPs (ensuring some node is usually very close to the device). Akamai is purported to have about 25 000 hosts distributed across 69 countries [2]. In a one-time pre-processing step, we can enumerate as many of their nodes as possible and then apply an existing virtual network coordinate system, Vivaldi [12], to assign them coordinates. The location-finding module chooses several nodes randomly out of this set, probes them to obtain round-trip times, then uses these values and the nodes' pre-computed virtual coordinates to determine the device's own virtual coordinates. Based on this, the module determines an additional set of landmarks that are close to it in virtual coordinate space and issues network measurements (pings and traceroutes) to these close landmarks. These measurements, in addition to the device's current internal- and external-facing IP addresses, are submitted to the core module as the current location information. After retrieval, this information can be used to geolocate the device, by potentially contacting the ISP hosting the edge routers.

**Putting it all together.** We describe the Adeona system in its entirety. A state of the client consists of the main cryptographic seed, the cache and its seed, and a time stamp. The main seed is used with an FSPRG to generate values associated to each state: the DHT indices, an encryption key, and an inter-update time. It also generates the next state's main seed and the next state's cache seed. The time stamp represents the time at which the current state should be used to send location information to the remote storage.

- (Initialization) The owner initializes the client by choosing random seeds and recording the time of initialization as the first state's time stamp. The cache is filled with random bits.
- (Active use) The main loop of the client proceeds as follows. The client, when executed, reads the current state and retrieves the current time (from, for example, the system clock). The client then transitions forward to the state that should be used to send the next update, based on the current time and the states' scheduled send times. The location cache uses its seed to appropriately encrypt each new location update received from the location module. At the scheduled send time, the main seed is used to generate several indices and an encryption key. The latter is used to encrypt the en-

tire cache. The result is inserted into OpenDHT under each index. The client then transitions to the next state. This means generating the next state's seed, the next state's cache seed, and the scheduled update time (the sum of the current update time and the inter-update delay). The old state data, except the cache, is erased.

- (Retrieval) To perform retrieval, the owner can use his or her copy of the initial state to recompute the sequence of states, their scheduled send times, and their associated indices and keys. From this information, he or she can determine the appropriate indices to search the remote storage (being careful to avoid indices from before the device went missing). After retrieving the caches, the owner can decrypt as described in Section 3.

## 5 Security Analysis

The Adeona system is designed to ensure location privacy, while retaining as much as possible the tracking abilities of solutions that provide weaker or no privacy properties. While we discuss other security evaluations and challenges inline in other sections, we treat here several key issues.

**Location privacy.** We discuss privacy first. We assume a privacy set of at least two participating devices, and do not consider omniscient adversaries that, in particular, can observe traffic at all locations visited by the device. (Such a powerful adversary can trivially compromise location privacy, assuming the device uses a persistent hardware MAC address.) The goal of adversaries is to use the Adeona system to learn more than their a priori knowledge about some device's visited locations. Because updates are anonymous and unlinkable, outsiders that see update traffic and the storage provider will not be able to associate the update to a device. The storage provider might associate updates that are later retrieved by the owner. This does not reveal anything about other updates sent by the owner's device. The randomized schedule obscures timing-related information that might otherwise reveal which device is communicating an update. Note also that the landmarks probed in our geolocationing module only learn that some device is probing them from an IP address. The thief cannot break the owner's location privacy due to our forward privacy guarantees.

Outsiders and the storage provider do learn that some device is at a certain location at a specific time (but not which device). Also, the number of devices currently using the system can be approximately determined (based on the rate of updates received), which could, for example, reveal a rough estimate of the number of devices behind a shared IP address. Moreover, these adversaries might attempt active attacks. For example, upon seeing an incoming update, the provider could immediately try to finger-print the source IP address [24]. Distributing the remote storage as with OpenDHT naturally makes such an attack more difficult to mount. There are also known preventative measures that mitigate a device's vulnerability to such attacks [24]. Finally, all of this could be protected against by sending updates via a system like Tor [14] (in deployment settings that would allow its use), which obfuscates the source IP address. See Section 8.4.

We remark that custom settings for Adeona's various parameters might reduce a device's privacy set. For example, if a client utilizes a cache size distinct from others, then this will serve to differentiate that client's updates. Likewise if a client submits more (or less) copies of each update to the remote storage, then the storage provider or outsiders might be able to differentiate its updates from those of other devices. Finally, a rate parameter significantly different from other clients' could allow tracking of the device.

**Device tracking.** We now discuss the goal of device tracking, which just means a system's ability to ensure updates about a missing device are retrieved by the owner. As mentioned previously, the goal here is for Adeona to engender the same tracking functionality as systems with weaker (or no) privacy guarantees. We therefore do not consider attacks which would also disable a normal tracking system: disabling the client, cutting off Internet access, destroying the device, etc. (Existing approaches to mitigating these attacks, like clever software engineering and/or hardware or BIOS support, are also applicable to our designs.) Nevertheless, Adeona as described in the previous section does have some limitations in this regard.

- OpenDHT does not provide everlasting persistence. This means that tracking fails for location updates more than a week old. Note that the location cache mechanism can be used to extend this time period. An alternate remote storage facility could also be used (see Section 7).
- Adeona schedules its updates at random times. If the device has Internet access for only a short time, this means that Adeona could miss a chance to send its update. We can trivially mitigate this by increasing the rate of our exponentially-distributed inter-update times (i.e., increase the frequency of updates), but at the cost of efficiency since this would mean sending more updates.
- The absolute privacy of retrieval relies on the device having a clock that the owner is loosely synchronized against. The client relies on the system clock to schedule updates. The thief could abuse this by,

for example, forcing the device's system clock to not progress. In the current implementation this would disrupt sending updates. Solutions for this are discussed in Section 8.1.

- Adeona relies on a stored state, and a thief could disable Adeona by tampering with it. For example, flipping even a single bit of the state will make all future updates unrecoverable. To ensure that the thief has to disable the client itself (and not just modify its state) we can use a tamper-evident FSPRG in conjunction with a "panic" mode of operation. See Sections 8.2 and 8.3.

For some of these bullets, we recall that many thieves will be unsophisticated. Therefore, in the common case the likelihood of the above attacks are small. (And, indeed, a sophisticated attacker could also compromise the tracking functionality of existing commercial, centralized alternatives.)

We also briefly mention that Adeona, like existing tracking systems, might not compose with some other mobile device security tools. For example, using a secure full-disk encryption system could render all software on the system unusable, including tracking software. We leave the question of how to securely combine tracking with other security mechanisms to future work.

Finally, while not a primary goal of our design, it turns out that Adeona's privacy mechanisms can actually *improve* tracking functionality. For one, the authentication of updates provided by our encryption mode means the owner knows that any received update was sent using the keys on the device, preventing in-transit tampering by outsiders or the storage provider. That updates are anonymous makes targeted Denial-of-Service attacks — in which the storage provider or an outsider attempts to selectively block or destroy an individual's updates — exceedingly difficult, if not impossible.

## 6 Implementation and Evaluation

To investigate the efficiency and practicality of our system, we have implemented several versions of the Adeona system as user-land applications for both Linux and Mac OS X. In all the versions, we used AES to implement the FSPRG. Encryption was performed using AES in counter mode and HMAC-SHA1 [3] in a standard Encrypt-then-MAC mode [4]. The OpenSSL crypto library[8] provided implementations of these primitives. We note that HMAC was used for convenience only; an implementation using AES for message authentication would also be straightforward. The rpcgen compiler was used to generate the client-side stubs for OpenDHT's put-get interface over the Sun RPC protocol. We also used Perl scripts to facilitate installation. We focus on three main versions.

- **adeona-0.2.1** implements the core functionality described in Sections 3 and 4. It uses the medium location-finding module of Section 4. The source code for adeona-0.2.1, not including the libraries mentioned above, consists of 7 091 lines of unoptimized C code. (Count includes comments and blank lines, i.e. calculated via `wc -l *.[ch]`.) This version is being readied for public release.

- **adeona-0.2.0** is a slightly earlier version of adeona-0.2.1 that differs in that it uses a simpler version of the forward-private location cache. Its cache only handles locations observed during scheduled updates (as opposed to more frequent checks for a change in location, meaning that locations could be missed if ill-timed). The source code for adeona-0.2.0 consists of 5 231 lines of unoptimized C code. This version was deployed in the field trial described in Section 6.3.

- **adeona-0.1** uses the same ciphertext cache mechanism as adeona-0.2.0, and additionally includes the tamper-evident FSPRG that will be described in Section 8.2, the panic mode that will be described in Section 8.3, and the full location-finding mechanism described in Section 4. The tamper-evident FSPRG is implemented using the signature scheme associated to the Boneh-Boyen identity-based encryption (IBE) scheme [7] and the anonymous IBE scheme is implemented using Boneh-Franklin [8] in a hybrid mode with the Encrypt-then-MAC scheme described above. The two schemes rely on the same underlying elliptic curves that admit efficiently computable bilinear pairings. It relies on the Stanford Pairings-Based Crypto (PBC) library version 0.4.11 [25] and specifically the "Type F" pairings. Not counting the PBC library, this version is implemented in 9 723 lines of C code.

The oldest version was mainly for experimenting with the extensions discussed in Section 8 and the new geolocation technique discussed in Section 4, while the newer two versions were largely re-writes to prepare for public use. The source code for any version is directly available from the authors.

### 6.1 Performance

We ran several benchmarks to gauge the performance of our design mechanisms. The system hosting the experiments was a dual-core 3.20 GHz Intel Pentium 4 processor with 1GB of RAM. It was connected to the Internet via a university network.

**Basic network operations.** Table 2 gives the Wall-clock time in milliseconds (calculated via the gettimeofday system call) to perform each basic network operation: an OpenDHT put of a 1024-byte payload, an OpenDHT

|  | Min | Mean | Median | Max | T/O |
|---|---|---|---|---|---|
| Put | 207 | 1 021 | 470 | 11 463 | 2 |
| Get | 2 | 240 | 77 | 11 238 | 3 |
| Loc medium | 5 642 | 13 270 | 15 531 | 30 381 | – |
| Loc full | 17 446 | 36 802 | 36 197 | 63 916 | – |

Table 1: Wall clock time in milliseconds/operation to perform basic network operations: DHT put, DHT get, a medium location-finding operation, and a full location-finding operation.

| **adeona-0.2.1** | $r = 0$ | $r = 10$ | $r = 100$ |
|---|---|---|---|
| Owner state | 75 | 75 | 75 |
| Client state (light) | 75 | 876 | 8 076 |
| Update (light) | 36 | 400 | 4000 |
| Client state (medium) | 75 | 27 116 | 270 476 |
| Update (medium) | 1 348 | 13 520 | 135 200 |

| **adeona-0.1** | $r = 0$ | $r = 10$ | $r = 100$ |
|---|---|---|---|
| Owner state | 3 544 | 3 545 | 3 548 |
| Client state (full) | 1 779 | 30 824 | 292 184 |
| Update (full) | 1 452 | 14 520 | 145 200 |

Table 2: Typical sizes in bytes of state and update data used by adeona-0.2.1 and adeona-0.1 on a 32-bit system, for different sizes of the ciphertext cache specified by $r$.

get of a 1024-byte payload, the time to do the 8 traceroutes used in the medium location-finding mechanism, and the time to do the full location-finding operation (as described in Section 4). Each operation was performed 100 times; shown is the min/mean/median/max time over the successful trials. The number of time outs (failures) for the put trials and get trials are shown in the column labeled T/O. The time out for OpenDHT RPC calls was set to 15 seconds in the implementation. For the location mechanisms, hop timeouts for traceroutes and timeouts for pings were set to 2 seconds (here an individual probe time out does not signify failure of the operation).

**Space utilization.** Table 2 details the space requirements in bytes of adeona-0.2.1 (adeona-0.2.0 has equivalent sizes) with light and medium location mechanisms and adeona-0.1 with the full location mechanism. Here, and below, the parameter $r$ specifies the size of the ciphertext cache used. When $r = 0$ this means that no cache was used (only the current location is inserted during an update). For ease-of-use (i.e., so one can print out or copy down state information) we encoded all persistently stored data in hex, meaning the sizes of stored state are roughly twice larger than absolutely necessary. The use of asymmetric primitives by adeona-0.1 for the tamper-evident FSPRG functionality and the IBE scheme account for its larger space utilization.

**Microbenchmarks.** Space constraints limit the amount of data we can report, and so our focus here is on adeona-0.1. It uses more expensive cryptographic primitives (elliptic curves supporting bilinear pairings), and we want to assess whether the extensions relying on them hinder performance significantly. Table 3 gives running times in milliseconds/operation for the basic operations used by adeona-0.1. (We omit the times for non-panic encryption, decryption, update, and retrieve; these times were at most those of the related panic-mode operations.) These benchmarks only used the light location-finding mechanism and each update was inserted to a single OpenDHT node. Each operation was timed for 100 repetitions both using the clock system call (the CPU columns) and gettimeofday (the Wall columns); shown is the min/mean/median/max time over the successful trials. Where applicable, the number of time outs (due to DHT operations) are shown in the column labeled T/O. Note that the retrieve operations only include retrieval for a single update. These benchmarks show that the extensions are not prohibitive: performance is dependent almost entirely on the speed of network operations.

## 6.2 Geolocation accuracy

As mentioned earlier, our system has been designed to convey various kinds of location information to the storage service. We then rely on previously proposed network measurement analysis techniques and/or database lookups to process the stored location information and derive a geographical estimate. The strengths and weaknesses of such techniques are well-documented. We therefore focus our evaluation on the active client-based measurement technique described in Section 4 that attempts to identify a set of nearby passive landmarks given a large number of geographically distributed landmarks.

First, we accumulated about 225 412 open DNS servers by querying Internet search engines for dictionary words and collecting the DNS servers which responded to lookups on the resulting hostnames. Next, we enumerated 8 643 Akamai nodes across the world by querying the DNS servers for the IP addresses of hostnames known to resolve to Akamai edge servers (e.g., `www.nba.com`). Finally, 50 PlanetLab [11] nodes were used as stand-ins for lost or stolen devices across the United States.

Having both targets and landmarks, we obtained round-trip time (RTT) measurements from the PlanetLab nodes to the passive Akamai servers. The PlanetLab nodes were able to obtain measurements to 6 200 of the Akamai servers on our list. We could then evaluate our geolocation technique by running it over these measurements. Figure 3 plots the cumulative distributions of our results and the RTT to the actual closest Akamai node. We also plot the cumulative distribution of the RTT to an Akamai node as given by a simple DNS lookup for 32 of our 50 targets (the other 18 nodes went down at

| Operation | | CPU | | | | Wall | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Median | Max | Min | Mean | Median | Max | T/O |
| Initialize core | | 210 | 329 | 330 | 460 | 215 | 367 | 348 | 1 082 | – |
| Verify FSPRG state | | 340 | 456 | 470 | 610 | 351 | 494 | 474 | 1 240 | – |
| Panic encryption | | 90 | 95 | 90 | 110 | 93 | 101 | 95 | 207 | – |
| Panic decryption | | 80 | 90 | 90 | 100 | 85 | 104 | 90 | 934 | – |
| Panic update | $r = 0$ | 440 | 559 | 570 | 700 | 612 | 1 653 | 977 | 15 347 | 9 |
| | $r = 10$ | 440 | 543 | 545 | 680 | 818 | 2 289 | 1 311 | 20 582 | 10 |
| | $r = 100$ | 540 | 666 | 690 | 800 | 2 953 | 12 599 | 7 439 | 165 950 | 5 |
| Panic retrieve | $r = 0$ | 80 | 89 | 90 | 100 | 92 | 499 | 207 | 12 003 | 7 |
| | $r = 10$ | 80 | 87 | 90 | 100 | 93 | 705 | 335 | 9 734 | 12 |
| | $r = 100$ | 80 | 87 | 90 | 100 | 116 | 2 458 | 1 555 | 21 734 | 5 |

Table 3: Time in milliseconds to perform basic operations in adeona-0.1.

the time of measurement). Our technique performs better than Akamai's own content delivery algorithms for more than 60% of the the targets we considered. In addition, we observe that it can find an Akamai server at most 7 milliseconds away.

## 6.3 Field trial

We conducted a small field trial to gain experience with our implementation of Adeona, reveal potential issues with our designs, and quantitatively gauge the efficacy of using OpenDHT as a remote storage facility. There were 11 participants each running the adeona-0.2.0 client with the same options: update rate parameter of 0.002 (about 7 updates an hour on average), location cache of size $r = 4$, and spatial replication of 4 (the core tries to insert each update to 4 DHT keys). The clients were instrumented to locally log all the location updates submitted over the course of the trial. At the end of the trial, we collected these client-side log files plus each owner's copy of the initial state, and used this data to attempt to retrieve a week's worth of updates[9] for each of the participants.

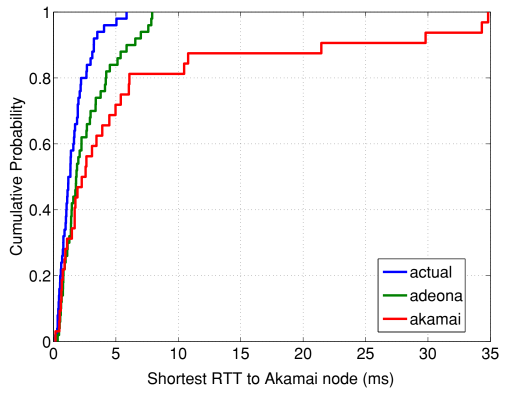
Results are shown in the left table of Figure 3. Here '# Inserts' refers to the total number of successful insertions into the DHT by the client in the week period. The 'Insert rate' column measures the fraction of these inserts that were retrieved. The '# Updates' column shows the total number of updates submitted by each client. Note that our replication mechanism means that each update causes the client to attempt 4 insertions of the location cache. The 'Update rate' column measures the percentage of location caches retrieved. As can be seen, this fraction is usually larger than the fraction of inserts retrieved, suggesting that replication across multiple DHT keys is beneficial. The 'Locations Found' column reports the number of unique locations (defined as distinct (internal IP, external IP) pair) found during retrieval versus reported. The final column measures the time, in minutes and seconds, that it took to perform retrieval for the user's updates for the whole week (note that we parallelized retrieval for each user).

We observed that OpenDHT may return "no data" for a key even when, in fact, there is data stored under that key. (This was detected when doing multiple get requests for a key.) Indeed, the failure to find two of the user locations was due to this phenomenon, and in fact repeating the retrieval operation found these locations as well.

## 7 Deployment Settings: Hardware Support and Dedicated Servers

In Section 2, we highlighted several settings for device tracking: internal corporate systems, third-party companies offering tracking services, and community-supported tracking for individuals. Each case differs in terms of what resources are available to both the tracking client and the remote storage. In Section 4 we built the Adeona system targeting a software client and OpenDHT repository, which works well for the third setting. Here we describe how our designs can work with other deployment scenarios.

A *hardware-supported* client can be deployed in several ways, including ASICs implementing client logic, trusted hardware modules (e.g., a TPM [35] or Intel's Active Management technology), or worked into existing system firmware components (e.g., a system BIOS). Hardware-support can be effectively used to ensure that the tracking client can only be disabled by the most sophisticated thieves and, possibly, that the client has access to a private and tamper-free state. However, targeting a system for use with a hardware-supported client *adds* to system requirements. While we do not work out all the (important) details of a hardware implementation of Adeona's client (leaving this to future work), we argue here that our techniques are amicable to this type of deployment. Adeona's core (Section 3) is particularly suited for implementation in hardware. It relies on a single cryptographic primitive, AES, which is highly optimized for both software and hardware. For example, recent research shows how to implement AES in only



| User | # Inserts | Insert Rate | # Updates | Update Rate | Locations Found | Retrieve Time |
|---|---|---|---|---|---|---|
| 01 | 491 | 0.89 | 251 | 0.94 | 11/12 | 12m 06s |
| 02 | 632 | 0.89 | 327 | 0.94 | 3/3 | 16m 04s |
| 03 | 622 | 0.84 | 321 | 0.91 | 2/2 | 17m 04s |
| 04 | 543 | 0.87 | 274 | 0.95 | 5/5 | 15m 03s |
| 05 | 617 | 0.88 | 309 | 0.96 | 4/4 | 19m 04s |
| 06 | 234 | 0.85 | 123 | 0.90 | 4/4 | 15m 06s |
| 07 | 359 | 0.89 | 199 | 0.95 | 5/6 | 18m 04s |
| 08 | 420 | 0.85 | 220 | 0.92 | 7/7 | 14m 06s |
| 09 | 504 | 0.91 | 259 | 0.97 | 1/1 | 11m 06s |
| 10 | 138 | 0.90 | 59 | 0.92 | 4/4 | 13m 04s |
| 11 | 302 | 0.81 | 175 | 0.91 | 6/6 | 14m 04s |

Figure 3: **(Left)** The cumulative distribution of the shortest RTT (in milliseconds) to an Akamai node found by Adeona compared to the actual closest Akamai node and Akamai's own content delivery algorithm. **(Right)** Field trial retrieval rates and retrieval times (in minutes and seconds).

3400 gates (on a "grain of sand") [15]. In its most basic form (without a location cache), the core only requires 16 bytes of persistent storage.

In settings where a third-party company offers tracking services, the design requirements are more relaxed compared to a community-supported approach. Particularly, such a company would typically offer *dedicated remote storage servers*. This would allow handling persistence issues server-side. Further, this kind of remote storage service is likely to provide better availability than DHTs, obviating the need to engineer the client to handle various kinds of service failures. Adeona is thus slightly over-engineered for this setting (e.g., we could dispense with the replication technique of Section 4). An interesting question that is raised in such a deployment setting is how to perform privacy-preserving access control. For obvious reasons, these remote storage facilities would want to restrict the parties able to insert data. If we use traditional authentication mechanisms, the authentication tag might reveal who is submitting the update. Thus one might think about using newer cryptographic primitives such as group or ring signatures [10, 31] that allow authentication while not revealing what member of a group is actually communicating the update.

Corporations or other large organizations might opt to *internally host storage facilities*, as per Scenario 5 of Section 2. Again, dedicated storage servers ease design constraints, meaning Adeona can be simplified for this setting. But there is again the issue of access control. (Though in this setting existing corporate VPN's, if these do not reveal the client's identity, might be used.) On the other hand, this kind of deployment raises other interesting questions. Particularly, the privacy set is necessarily restricted to only employees of the corporation, and so an adversary might be able to aggregate information about overall employee location habits even if the adversary cannot track individual employees.

## 8 Extensions

We describe several extensions for the Adeona system that highlight its versatility and extensibility. These include: removing the reliance on synchronized clocks, tamper-evident FSPRGs for untrusted local storage, a panic mode of operation that does not rely on state, the use of anonymous channels, and enabling communication from an owner back to a lost device.

### 8.1 Avoiding synchronized time

The Adeona system, as described in Section 4, utilizes a shared clock between the client and owner to ensure safe retrieval. This is realized straightforwardly if the client is loosely synchronized against an external clock (e.g., via NTP [27]). In deployment scenarios where the device cannot be guaranteed to maintain synchronization or the thief might maliciously modify the system clock, we can modify the client and retrieval process as follows.

Whenever the client is executed, it reads the current state (which is now just the current cryptographic seed for the FSPRG and the cache) and computes the inter-update time $\delta$ associated to the state. It then waits that amount of time before sending the next update and progressing the state. For retrieval, the owner can still compute all of the inter-update times, and use these to estimate when a state was used to send an update. If the client runs continuously from initialization, then a state will be used when predicted by the sum of the inter-update times of earlier states. If the client is not run continuously from initialization, then a state might be used to send an update *later* (relative to absolute time) than predicted by the inter-update times. It is therefore privacy-preserving for the owner to retrieve any states estimated to be sent after the time at which the device was lost. The owner might also query prior states to search for relevant updates, but being careful not to go too far back (lest he begin querying for updates sent before the device was lost).

## 8.2 Detection of client state tampering

The Adeona system relies on the client's state remaining unmodified. Compared to a (hypothetical) stateless client, this allows a new avenue for disabling the device. To rectify this disparity between the ideal (in which an adversary has to disable/modify the client executable) and Adeona we design a novel cryptographic primitive, a *tamper-evident FSPRG*, that allows cryptographic validation of state. By adding this functionality to Adeona we remove this avenue for disabling tracking functionality. Moreover, we believe that tamper-evident FSPRGs are likely of independent interest and might find use in further contexts where untrusted storage is in use, e.g., when the Adeona core is implemented in hardware but the state is stored in memory accessible to an adversary.

A straightforward construction would work as follows. The owner, during initialization, also generates a signing key and a verification key for a digital signature scheme. Then, the initialization routine generates the core's values $s_i, c_{i,1}, T_i$ for each future state $i$ that could be used by the client, and signs these values. The verification key and resulting signatures are placed in the client's storage, along with the normal initial state. The client, to validate lack of tampering with FSPRG states, can verify the state's $s_i, c_{i,1}, T_i$ values via the digital signature's verification algorithm and the (stored) verification key. Unfortunately this approach requires a large amount of storage (linear in the number of updates that could be sent). Moreover, a very sophisticated thief could just mount a replacement attack: substitute his or her own state, verification key, and signatures for the owner's. Note this attack does not require modifying or otherwise interfering with the client executable. We can do better on both accounts.

To stop replacement attacks, we can use a trusted authority to generate a certificate for the owner's verification key (which should also tie it to the device). Then the trusted authority's verification key can be hard-coded in the client executable and be used to validate the owner's (stored) verification key. To reduce the storage space required, we have the owner, during initialization, only sign the *final* state's values. To verify, the client can seek forward with the FSPRG (without yet deleting the current state) to the final state and then verify the signature. (A counter can be used to denote how many states the client needs to progress to reach the final one.) Under reasonable assumptions regarding the FSPRG (in particular, that it's difficult to find two distinct states that lead to the same future state) and the assumption that the digital signature scheme is secure, no adversary will be able to generate a state that deviates from the normal progression, yet verifies. A clever thief might try to roll the FSPRG forward in the normal progression, to cause a long wait before the next update. This can be defended against with a straightforward check relative to the current time: if the next update is too far away, then assume adversarial modification. We could also store the signatures of some fraction of the intermediate states in order to operate at different points of a space/computation trade-off.

## 8.3 Private updates with tampered state

If the client detects tampering with the state, then it can enter into a "panic" mode which does not rely on the stored state to send updates. One might have panic mode just send updates in the clear (because these locations are presumably not associated with the owner), but there can be reasons not to do this. For example, configuration errors by an owner could mistakenly invoke panic mode.

Panic mode can still provide some protection for updates without relying on shared state, as follows. We assume the client and owner have access to an immutable, unique identification string $\mathsf{ID}$. In practice this $\mathsf{ID}$ could be the laptop's MAC address. We also use a cryptographic hash function $H\colon \{0,1\}^* \to \{0,1\}^h$, such as SHA-256 for which $h = 256$ bits. Then pick a parameter $b \in [0..h]$. For each update, the client generates a sequence of indexes via $I_1 = H(1 \,||\, T \,||\, H(\mathsf{ID})|_b)$, $I_2 = H(2 \,||\, T \,||\, H(\mathsf{ID})|_b)$, etc. Here $T$ is the current date and $H(\mathsf{ID})|_b$ denotes hashing $\mathsf{ID}$ and then taking the first $b$ bits of the result. (Varying the parameter $b$ enables a simple time-privacy trade off known as "bucketization".)

Location information can be encrypted using an anonymous identity-based encryption scheme [8]. Using a trusted key distribution center, each owner can receive a secret key associated to their device's $\mathsf{ID}$. (Note that the center will be able to decrypt updates, also.) Encryption to the owner only requires $\mathsf{ID}$. This is useful because then encryption does not require stored per-device state, under the presumption that $\mathsf{ID}$ is always accessible. The ciphertext can be submitted under the indices. The owner can retrieve these panic-mode updates by re-computing the indices using $\mathsf{ID}$ and the appropriate dates and then using trial decryption.

## 8.4 Anonymous channels

Systems such as Tor [14] implement anonymous channels, which can be used to effectively obfuscate from recipients the originating IP address of Internet traffic. The Adeona design can easily compose with any such system by transmitting location updates to the remote storage across the anonymous channel. The combination of Adeona with an anonymous routing system provides several nice benefits. It means that the storage provider and outsiders do not trivially see the originating IP address, meaning active fingerprinting attacks are prevented. Additionally, it merges the anonymity set of Adeona with that of the anonymous channel system. For example, even if there exists only a single user of Adeona, that user might nevertheless achieve some degree of location privacy using anonymous channels.

On the other hand, attempting to use anonymous channels without Adeona does not satisfy our system goals. The now more complex clients would not necessarily be suitable for some deployment settings (e.g. hardware implementations). It would force a reliance on a complex, distributed infrastructure in all deployment settings. This reliance is particularly bad in the corporate setting. Routing location updates through nodes not controlled by the company could actually decrease corporate privacy: outsiders could potentially learn employee locations (e.g., see [36]). Moreover, when analyzing how to utilize anonymous channels and meet our tracking and privacy goals, it is easy to see that even with the anonymous channel one still benefits from Adeona's mechanisms. Imagine a hypothetical system based on anonymous channels. Because the storage provider is potentially adversarial, the system would still need to encrypt location information and so also provide an index to enable efficient search of the remote storage. Because the source IP is hidden, one might utilize a static, anonymous identifier. This would allow the storage provider to, at the very least, link update times to a single device, which leaks more information than if the indices are unlinkable.

## 8.5 Sending commands to the device

In situations where a device is lost, an owner might wish to not only retrieve updates from it, but also securely send commands back to it. For example, such a channel would allow remotely deleting sensitive data. We can securely instantiate a full duplex channel using the remote storage as a bulletin board. An owner could post encrypted and signed messages to the remote storage under indices of future updates. The client, during an update, would first do a retrieve on the indices to be used for the update, thereby receiving the encrypted and authenticated commands. Standard encryption and authentication tools can be used, including using cryptographic keys derived from the FSPRG seed in use on the client. In terms of location privacy, the storage provider would now additionally learn that two entities are communicating via the bulletin board, but not which entities.

## 9 Conclusion

This paper develops mechanisms by which one can build *privacy-preserving device tracking systems*. These systems simultaneously hide a device owner's legitimately visited locations from powerful adversaries, while enabling tracking of the device after it goes missing. Moreover, we do so while using third party services that are not trusted in terms of location privacy. Our mechanisms are efficient and practical to deploy. Our client-side mechanisms are well-suited for hardware implementations. This illustrates that not only can one circumvent a trade-off between security and privacy, but one can do so in practice for real systems.

We implemented Adeona, a full privacy-preserving tracking system based on OpenDHT that allows for immediate, community-orientated deployment. Its core module, the cryptographic engine that renders location updates anonymous and unlinkable, can be easily used in further deployment settings. To evaluate Adeona, we ran a field trial to gain experience with a deployment on real user's systems. Our conclusion is that our approach is sound and an immediately viable alternative to tracking systems that offer less (or no) privacy guarantees. Lastly, we also presented numerous extensions to Adeona that address a range of issues: disparate deployment settings, increased functionality, and improved security. The techniques involved, particularly our tamper-evident FSPRG, are likely of independent interest.

## Notes

[1]EmailMe is a fictional system, though its functionality is based on products such as PC Phone Home [9] and Inspice [21].

[2]A flea market is a type of ad-hoc market where transactions are typically anonymous and done in cash.

[3]AllDevRec is a fictional company, though the services it offers are comparable to those advertised by Absolute Software [1], which has tracking software pre-installed in the BIOS of some Dell laptops.

[4]A real example of such insider abuse is found in [20].

[5]The owner could download the entire database and do trial decryption, but with many users this would be prohibitively expensive.

[6]One could also utilize as basic primitive a keyed hash function.

[7]To be precise, the guarantee is that OpenDHT guarantees not to expire a key-value pair before its time-to-live passes, barring some catastrophic failure of the DHT service [30].

[8]Systems we built on had version 0.9.7l or later. We used SHA1, instead of the more secure SHA-256, due to its lack of implementation in OpenSSL 0.9.7l (the most recent version available for OSX).

[9]To be precise, the search was for any update potentially sent over the course of 6 days and 23 hours. The final hour was omitted for simplicity since it avoided retrieving updates being expired by OpenDHT.

## References

[1] Absolute Software. Computrace LoJack for Laptops. http://www.absolute.com/solutions-theft-recovery.asp

[2] Akamai website. http://www.akamai.com/

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. *CRYPTO*, 1996.

[4] M. Bellare and C. Namprempre. Authenticated-Encryption: Relations among notions and analysis of the generic composition paradigm. *ASIACRYPT*, 2000.

[5] M. Bellare and B. Yee. Forward-Security in Privatey-Key Cryptography. *CT-RSA*, 2003.

[6] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, vol. 13, no. 4, pp. 850–864, 1984.

[7] D. Boneh and X. Boyen. Efficient selective-ID secure identity based encryption without random oracles. In EUROCRYPT, 2004.

[8] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, 2001.

[9] Brigadoon Software, Inc. PC Phone Home. http://www.pcphonehome.com/

[10] D. Chaum and E. van Heyst. Group signatures. In *EUROCRYPT*, 1991.

[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An overlay testbed for broad-coverage services *ACM SIGCOMM*, 2003.

[12] F. Dabek, R. Cox, F. Kaashoek, and R. Morris Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[13] L. Devroye. Non-Uniform Random Variate Generation. New York, Springer-Verlag, 1986.

[14] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. *USENIX Security Symposium*, USENIX, vol. 13, pp. 9–13, 2004.

[15] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES implementation on a grain of sand. *IEE Proc. Inf. Secur.*, vol. 152, no. 1, pp. 13–20, Oct. 2005.

[16] L. Gordon, M. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI Computer Crime and Security Survey 2006. Computer Security Institute publications, 2006.

[17] M. Gruteser and D. Grunwald. A Methodological Assessment of Location Privacy Risks in Wireless Hotspot Networks *Security in Pervasive Computing – SPC*, 2003.

[18] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida. Constraint-based geolocation of Internet hosts. *To appear in ACM Transactions on Networking.*

[19] IEEE Standards Association. IEEE Std 802.11i-2004. 2004.

[20] InformationWeek. Federal Agent Indicted for Using Homeland Security Database To Stalk Girlfriend. http://www.informationweek.com/shared/printableArticle.jhtml?articleID=201807903

[21] Inspice. Inspice Trace. http://www.inspice.com/

[22] M. Jakobsson and S. Wetzel. Security Weaknesses in Bluetooth. CT-RSA, 2001.

[23] A. Juels. RFID Security and Privacy: A Research Survey, *IEEE Journal on Selected Areas in Communications*, 2006.

[24] T. Kohno, A. Broido, and K.C. Claffy. Remote physical device fingerprinting. *IEEE Symposium on Security and Privacy*, 2005.

[25] B. Lynn. Stanford Pairings-Based Crypto Library. http://crypto.stanford.edu/pbc/.

[26] D. McGrew and J. Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. *INDOCRYPT*, 2004.

[27] D. Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *IEEE/ACM Trans. Netw.*, vol. 3 no. 3, pp. 245–254, 1995.

[28] New York Times. At U.S. Borders, Laptops Have No Right to Privacy. http://www.nytimes.com/2006/10/24/business/24road.html . October 24, 2006.

[29] Raytheon Oakley Systems. SureFind. http://www.oakleynetworks.com/products/surefind.php

[30] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*, 2005.

[31] R. Rivest, A. Shamir, and Y. Tauman How to leak a secret. In *ASIACRYPT*, 2001.

[32] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. ACM Transactions on Information and System Security, vol. 1, no. 3, 1999.

[33] Slashdot. US Courts Consider Legality of Laptop Inspection. http://yro.slashdot.org/article.pl?sid=08/01/08/1641209&tid=158 . January 8, 2008.

[34] Tri-8, Inc. MyLaptopGPS. http://www.mylaptopgps.com/

[35] Trusted Computing Group. http://www.trustedcomputinggroup.org/specs/TPM/

[36] Wired. Rogue Nodes Turn Tor Anonymizer Into Eavesdropper's Paradise. http://www.wired.com/politics/security/news/2007/09/embassy_hacks . September 10, 2007.

[37] XTool Mobile Security, Inc. XTool Laptop Tracker. http://www.xtool.com/xtooltracker.aspx

[38] zTrace Technologies. http://www.ztrace.com/

# Panalyst: Privacy-Aware Remote Error Analysis on Commodity Software

*Rui Wang*[†], *XiaoFeng Wang*[†] *and Zhuowei Li*[‡]
[†]*Indiana University at Bloomington,* [‡]*Center for Software Excellence, Microsoft*
{*wang63,xw7*}*@indiana.edu, zhuowei.li@microsoft.com*

## Abstract

Remote error analysis aims at timely detection and remedy of software vulnerabilities through analyzing runtime errors that occur on the client. This objective can only be achieved by offering users effective protection of their private information and minimizing the performance impact of the analysis on their systems without undermining the amount of information the server can access for understanding errors. To this end, we propose in the paper a new technique for privacy-aware remote analysis, called *Panalyst*. Panalyst includes a client component and a server component. Once a runtime exception happens to an application, Panalyst client sends the server an initial error report that includes only public information regarding the error, such as the length of the packet that triggers the exception. Using an input built from the report, Panalyst server performs a taint analysis and symbolic execution on the application, and adjusts the input by querying the client about the information upon which the execution of the application depends. The client agrees to answer only when the reply does not give away too much user information. In this way, an input that reproduces the error can be gradually built on the server under the client's consent. Our experimental study of this technique demonstrates that it exposes a very small amount of user information, introduces negligible overheads to the client and enables the server to effectively analyze an error.

## 1 Introduction

Remote analysis of program runtime errors enables timely discovery and patching of software bugs, and has therefore become an important means to improve software security and reliability. As an example, Microsoft is reported to fix 29 percent of all Windows XP bugs within Service Pack 1 through its Windows Error Reporting (WER) utility [20]. Remote error analysis is typically achieved by running an error reporting tool on a client system, which gathers data related to an application's runtime exception (such as a crash) and transmits them to a server for diagnosis of the underlying software flaws. This paradigm has been widely adopted by software manufacturers. For example, Microsoft relies on WER to collect data should a crash happen to an application. Similar tools developed by the third party are also extensively used. An example is BugToaster [27], a free crash analysis tool that queries a central database using the attributes extracted from a crash to seek a potential fix. These tools, once complemented by automatic analysis mechanisms [44, 34] on the server side, will also contribute to quick detection and remedy of critical security flaws that can be exploited to launch a large-scale cyber attack such as Worm epidemic [47, 30].

The primary concern of remote error analysis is its privacy impact. An error report may include private user information such as a user's name and the data she submitted to a website [9]. To reduce information leaks, error reporting systems usually only collect a small amount of information related to an error, for example, a snippet of the memory around a corrupted pointer. This treatment, however, does not sufficiently address the privacy concern, as the snippet may still carry confidential data. Moreover, it can also make an error report less informative for the purpose of rapid detection of the causal bugs, some of which could be security critical. To mitigate this problem, prior research proposes to instrument an application to log its runtime operations and submit the sanitized log once an exception happens [25, 36]. Such approaches affect the performance of an application even when it works normally, and require nontrivial changes to the application's code: for example, Scrash [25] needs to do source-code transformation, which makes it unsuitable for debugging commodity software. In addition, these approaches still cannot ensure that sufficient information is gathered for a quick identification of critical security flaws. Alternatively, one can analyze a vulnerable program directly on the client [29]. This involves intensive debugging operations such as replaying the input that causes a crash and analyzing an executable at the instruction level [29], which could be too intrusive to the user's normal operations to be acceptable for a practical deployment. Another problem is that such an analysis consumes a large amount of computing resources. For example, instruction-level tracing of program execution usually produces an execution trace of hundreds of megabytes [23]. This can hardly be afforded by the client with limited resources, such as Pocket PC or PDA.

We believe that a good remote analyzer should help the user effectively control the information to be used in an error diagnosis, and avoid expensive operations on the client side and modification of an application's source or binary code. On the other hand, it is also expected to offer sufficient information for automatic detection and remedy of critical security flaws. To this end, we propose Panalyst, a new technique for privacy-aware remote analysis of the crashes triggered by network inputs. Panalyst aims at automatically generating a new input on the server side to accurately reproduce a crash that happens on the client, using the information disclosed according to the user's privacy policies. This is achieved through collaboration between its client component and server component. When an application crashes, Panalyst client identifies the packet that triggers the exception and generates an initial error report containing nothing but the public attributes of the packet, such as its length. Taking the report as a "taint" source, Panalyst server performs an instruction-level taint analysis of the vulnerable application. During this process, the server may ask the client questions related to the content of the packet, for example, whether a tainted branching condition is true. The client answers the questions only if the amount of information leaked by its answer is permitted by the privacy policies. The client's answers are used by the server to build a new packet that causes the same exception to the application, and determine the property of the underlying bug, particularly whether it is security critical.

Panalyst client measures the information leaks associated with individual questions using *entropy*. Our privacy policies use this measure to define the maximal amount of information that can be revealed for individual fields of an application-level protocol. This treatment enables the user to effectively control her information during error reporting. Panalyst client does not perform any intensive debugging operations and therefore incurs only negligible overheads. It works on commodity applications without modifying their code. These properties make a practical deployment of our technique plausible. In the meantime, our approach can effectively analyze a vulnerable application and capture the bugs that are exploitable by malicious inputs. Panalyst can be used by software manufacturers to demonstrate their "due diligence" in preserving their customers' privacy, and by a third party to facilitate collaborative diagnosis of vulnerable software.

We sketch the contributions of this paper as follows:

- *Novel framework for remote error analysis.* We propose a new framework for remote error analysis. The framework minimizes the impact of an analysis to the client's performance and resources, lets the user maintain a full control of her information, and in the meantime provides her the convenience to contribute to the analysis the maximal amount of information she is willing to reveal. On the server side, our approach interleaves the construction of an accurate input for triggering an error, which is achieved through interactions with the client, and the analysis of the bug that causes the error. This feature allows our analyzer to make full use of the information provided by the client: even if such information is insufficient for reproducing the error, it helps discover part of input attributes, which can be fed into other debugging mechanisms such as fuzz testing [35] to identify the bug.

- *Automatic control of information leaks.* We present our design of new privacy policies to define the maximal amount of information that can be leaked for individual fields of an application-level protocol. We also developed a new technique to enforce such policies, which automatically evaluates the information leaks caused by responding to a question and then makes decision on whether to submit the answer in accordance with the policies.

- *Implementation and evaluations.* We implemented a prototype system of Panalyst and evaluated it using real applications. Our experimental study shows that Panalyst can accurately restore the causal input of an error without leaking out too much user information. Moreover, our technique has been demonstrated to introduce nothing but negligible overheads to the client.

The rest of the paper is organized as follows. Section 2 formally models the problem of remote error analysis. Section 3 elaborates the design of Panalyst. Section 4 describes the implementation of our prototype system. Section 5 reports an empirical study of our technique using the prototype. Section 6 discusses the limitations of our current design. Section 7 presents the related prior research, and Section 8 concludes the paper and envisions the future research.

## 2 Problem Description

We formally model the problem of remote error analysis as follows. Let $P : S \times I \rightarrow S$ be a program that maps an initial process state $s \in S$ and an input $i \in I$ to an end state. A state here describes the data in memory, disk and register that are related to the process of $P$. A subset of $S$, $E_b$, contains all possible states the process can end at after an input exploits a bug $b$.

Once $P$ terminates at an error state, the client runs an error reporting program $G : I \rightarrow R$ to generate a report $r \in R$ for analyzing $P$ on the server. The report must be created under the constraints of the computing resources the client is able or willing to commit. Specifically, $C_t : \{G\} \times I \times R \rightarrow \Re$ measures the delay experienced by the user during report generation, $C_s : \{G\} \times I \times R \rightarrow \Re$ measures the storage overhead, and $C_n : \{G\} \times I \times R \longrightarrow \Re$ measures the bandwidth used for transmitting the report. To produce and submit a report $r \in R$, the computation time, storage consumption and bandwidth usage must be bounded by certain thresholds: formally, $(C_t(G, i, r) \leq Th_t) \wedge (C_s(G, i, r) \leq Th_s) \wedge (C_w(G, i, r) \leq Th_w)$, where $Th_t$, $Th_s$ and $Th_w$ represent the thresholds for time, storage space and bandwidth respectively. In addition, $r$ is allowed to be submitted only when the amount of information it carries is acceptable to the user. This is enforced using a function $L : R \times I \rightarrow \Re$ that quantifies the information leaked out by $r$, and a threshold $Th_l$. Formally, we require $L(r, i) \leq Th_l$.

The server runs an analyzer $D : R \rightarrow I$ to diagnose the vulnerable program $P$. $D$ constructs a new input using $r$ to exploit the same bug that causes the error on the client. Formally, given $P(i) \in E_b$ and $r = G(i)$, the analyzer identifies another input $i'$ from $r$ such that $P(i') \in E_b$. This is also subject to resource constraints. Specifically, let $C'_t : \{D\} \times R \times I \rightarrow \Re$ be a function that measures the computation time for running $D$ and $C'_s : \{D\} \times R \times I \rightarrow \Re$ that measures the storage overhead. We have: $(C'_t(D, r, i') \leq Th'_t) \wedge (C'_s(D, r, i') \leq Th'_s)$, where $Th'_t$ and $Th'_s$ are the server's thresholds for time and space respectively.

A solution to the above problem is expected to achieve three objectives:

- *Low client overheads.* A practical solution should work effectively under very small $Th_t$, $Th_s$ and $Th_w$. Remote error analysis aims at timely detecting critical security flaws, which can only be achieved when most clients are willing to collaborate in most of the time. However, this will not happen unless the client-side operations are extremely lightweight, as clients may have limited resources and their workloads may vary with time. Actually, customers could be very sensitive to the overheads brought in by error reporting systems. For example, advice has been given to turn off WER on Windows Vista and Windows Mobile to improve their performance [12, 17, 13]. Therefore, it is imaginable that many may stop participating in error analysis in response to even a slight increase of overheads. As a result, the chance to catch dangerous bugs can be significantly reduced.
- *Control of information leaks.* The user needs to have a full control of her information during an error analysis. Otherwise, she may choose not to participate. Indispensable to this objective is a well-constructed function $L$ that offers the user a reasonable measure of the information within an error report. In addition, privacy policies built upon $L$ and a well-designed policy enforcer will automate the information control, thereby offering the user a reliable and convenient way to protect her privacy.
- *Usability of error report.* Error reports submitted by the user should contain ample information to allow a new input $i'$ to be generated within a short period of time (small $Th'_t$) and at a reasonable storage overhead (small $Th'_s$). The reports produced by the existing systems include little information, for example, a snapshot of the memory around a corrupted pointer. As a result, an analyzer may need to exhaustively explore a vulnerable program's branches to identify the bug that causes the error. This process can be very time-consuming. To improve this situation, it is important to have a report that gives a detailed description about how an exploit happens.

In Section 3, we present an approach that achieves these objectives.

## 3 Our Approach

In this section, we first present an overview of Panalyst and then elaborate on the designs of its individual components.

### 3.1 Overview

Panalyst has two components, client and server. Panalyst client logs the packets an application receives, notifies the server of its runtime error, and helps the server analyze the error by responding to its questions as long as the answers are permitted by the user's privacy policies. Panalyst server runs an instruction-level taint analysis on the application's executable using an empty input, and evaluates the execution symbolically [37] in the meantime. Whenever the server encounters a tainted value that affects the choice of execution paths or memory access,

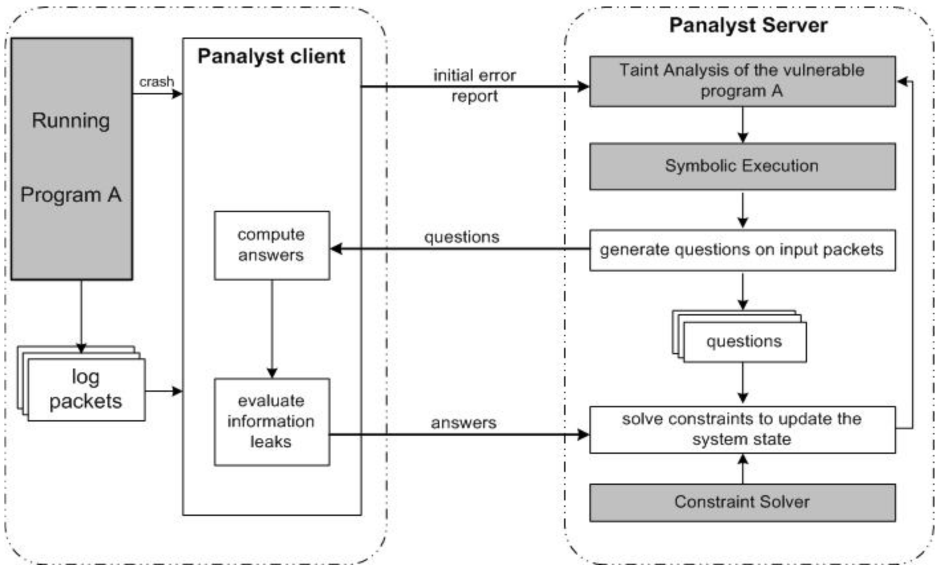


Figure 1: The Design of Panalyst.

it queries the client using the symbolic expression of that value. From the client's answer, the server uses a constraint solver to compute the values of the input bytes that taint the expression. We illustrate the design of our approach in Figure 1.

```
1   If(strcmp(conn->requestMethod,  "POST" ) == 0){
2       buf = malloc(conn->ContentLength+1024);
3       for(len=0;;) {
4           recv(sd, buf+len, 1, 0);
5           len++;
6           if(buf[len-1] ==  '\n' ) break;
7       }
8   }
```

Figure 2: An Illustrative Example.

**An example.** Here we explain how Panalyst works through an example, a program described in Figure 2. The example is a simplified version of Null-HTTPd [8]. It is written in `C` for illustration purpose: Panalyst actually is designed to work on binary executables. The program first checks whether a packet is an HTTP `POST` request. If so, it allocates a buffer with the size computed by adding 1024 to an integer derived from the `Content-Length` field and moves the content of the request to that buffer. A problem here is that a buffer overflow can happen if `Content-Length` is set to be negative, which makes the buffer smaller than expected. When this happens, the program may crash as a result of writing to an illegal address or being terminated by an error detection mechanism such as `GLIBC` error detection.

Panalyst client logs the packets recently received by the program. In response to a crash, the client identifies the packet being processed and notifies Panalyst server of the error. The server then starts analyzing the vulnerable program at instruction level using an empty HTTP request as a taint source. The request is also described by a set of symbols, which the server uses to compute a symbolic expression for the value of every tainted memory location or register. When the execution of the program reaches Line 1 in Figure 2, the values of the first four bytes on the request need to be revealed so as to determine the branch the execution should follow. For this purpose, the server sends the client a question: "$B_1B_2B_3B_4$ = '`POST`'?", where $B_j$ represents the $jth$ byte on the request. The client checks its privacy policies, which defines the maximal number of bits of information allowed to be leaked for individual HTTP field. In this case, the client is permitted to reveal the keyword `POST` that is deemed nonsensitive. The server then fills the empty request with these letters and moves on to the branch determined by the client's answer. The instruction on Line 2 calls `malloc`. The function accesses memory using a pointer built upon the content of `Content-Length`, which is tainted. To enable this memory access, the server sends the symbolic expression of the pointer to the client to query its concrete value. The client's reply allows the server to add more bytes to the request it is working on. Finally, the execution hits Line 3, a loop to move request content to the buffer allocated through `malloc`. The loop is identified by the server from its repeated instruction pattern. Then, a question is delivered to the client to query its exit condition: " where is the first byte $B_j$ = '$\backslash n$'?". This question concerns request content, a field on which the privacy policies forbid the client to leak out more than certain amount of information. Suppose that threshold is 5 bytes. To answer the question, only one byte needs to be given away: the position of the byte '$\backslash n$'. Therefore, the client answers the question, which enables the server to construct a new packet to reproduce the crash.

The performance of an analysis can be improved by sending the server an initial report with all the fields that are deemed nonsensitive according the user's privacy policies. In the example, these fields include keywords such as '`POST`' and the `Content-Length` field. This treatment reduces the communication overheads during an analysis.

**Threat model.** We assume that the user trusts the information provided by the server but does not trust her data with the server. The rationale behind this assumption is based upon the following observations. The owners of the server are often software manufacturers, who have little incentive to steal their customers' information. What the user does not trust is the way in which those parties manage her data, as improper management of the data can result in leaks of her private information. Actually, the same issue is also of concern to those owners, as they could be reluctant to take the liability for protecting user information. Therefore, the client can view the server as a benign though unreliable partner, and take advantage of the information it discovers from the vulnerable program to identify sensitive data, which we elaborate in Section 3.2.

Note that this assumption *is not* fundamental to Panalyst: more often than not, the client is capable of identifying sensitive data on its own. As an example, the aforementioned analysis on the program in Figure 2 does not rely on any trust in the server. Actually, the assumption only serves an approach for defining fine-grained privacy policies in our research (Section 3.2), and elimination of the assumption, though may lead to coarser-grained policies under some circumstances, will not invalidate the whole approach.

## 3.2 Panalyst Client

Panalyst client is designed to work on the computing devices with various resource constraints. Therefore, it needs to be extremely lightweight. The client also includes a set of policies for protecting the user's privacy and a mechanism to enforce them. We elaborate such a design as follows.

**Packet logging and error reporting.** Panalyst client intercepts the packets received by an application, extracts their application-level payloads and saves them to a log file. This can be achieved either through capturing packets at network layer using a sniffer such as Wireshark [1], or by interposing on the calls for receiving data from network. We chose the latter for prototyping the client: in our implementation, an application's socket calls are intercepted using `ptrace` [10] to dump the application-level data to a log. The size of the file is bounded, and therefore only the most recent packets are kept.

When a serious runtime error happens, the process of a vulnerable program may crash, which triggers our error analysis mechanism. Runtime errors can also be detected by the mechanisms such as `GLIBC` error detection, Windows build-in diagnostics [11] or other runtime error detection techniques [28, 21]. Once an error happens to an application, Panalyst client identifies the packets it is working on. This is achieved in our design by looking at all the packets within one TCP connection. Specifically, the client marks the beginning of a connection once observing an `accept` call from the application and the end of the connection when it detects `close`. After an exception happens, the client concatenates the application-level payloads of all the packets within the current connection to form a *message*, which it uses to talk to the server. For simplicity, our current design focuses on the error triggered by network input and assumes that all information related to the exploit is present in a single connection. Panalyst can be extended to handle the errors caused by other inputs such as data from a local file through logging and analyzing these inputs. It could also work on multiple connections with the support of the state-of-art replay techniques [43, 32] that are capable of replaying the whole application-layer session to the vulnerable application on the server side. When a runtime error occurs, Panalyst client notifies the server of the type of the error, for example, segmentation fault and illegal instruction. Moreover, the client can ship to the server part of the message responsible for the error, given such information is deemed nonsensitive according to the user's privacy policies.

After reporting to the server a runtime error, Panalyst client starts listening to a port to wait for the questions from the server. Panalyst server may ask two types of questions, either related to a tainted branching condition or a tainted pointer a vulnerable program uses to access memory. In the first case, the client is supposed to answer "yes" or "no" to the question described by a symbolic inequality: $C(B_{k[1]}, \ldots, B_{k[m]}) \leq 0$, where $B_{k[j]}$ ($1 \leq j \leq m$) is the symbol for the $k[j]$th byte on the causal message. In the second case, the client is queried about the concrete value of a symbolic pointer $S(B_{k[1]}, \ldots, B_{k[m]})$. These questions can be easily addressed by the client using the values of these bytes on the message. However, the answers can be delivered to the server only after they are checked against the user's privacy policies, which we describe below.

**Privacy policies.** Privacy policies here are designed to specify the maximal amount of information that can be given away during an error analysis. Therefore, they must be built upon a proper measure of information. Here, we adopt *entropy* [48], a classic concept of information theory, as the measure. Entropy quantifies uncertainty as number of bits. Specifically, suppose that an application field $A$ is equally likely to take one of $m$ different values. The entropy of $A$ is computed as $\log_2 m$ bits. If the client reveals that $A$ makes a path condition true, which reduces the possible values the field can have to a proportion $\rho$ of $m$, the exposed information is quantified as: $\log_2 m - \log_2 \rho m = -\log_2 \rho$ bits.

The privacy policies used in Panalyst define the maximal number of bytes of the information within a protocol field that can be leaked out. The number here is called *leakage threshold*. Formally, denote the leakage threshold for a field $A$ by $\tau$. Suppose the server can infer from the client's answers that $A$ can take a proportion $\rho$ of all possible values of that field. The privacy policy requires that the following hold: $-\log_2 \rho \leq \tau$. For example, a policy can specify that no more than 2 bytes of the URL information within an HTTP request can be revealed to the server. This policy design can achieve a fine-grained control of information. As an example, let us consider HTTP requests: protocol keywords such as `GET` and `POST` are usually deemed nonsensitive, and therefore can be directly revealed to the server; on the other hand, the URL field and the cookie field can be sensitive, and need to be protected by low leakage thresholds. Panalyst client includes a protocol parser to partition a protocol message into fields. The parser does not need to be precise: if it cannot tell two fields apart, it just treats them as a single field.

A problem here is that applications may use closed protocols such as ICQ and SMB whose specifications are not publically available. For these protocols, the whole protocol message has to be treated as a single field, which unfortunately greatly reduces the granularity of control privacy policies can have. A solution to this problem is to partition information using the parameters of API (such as Linux kernel API, `GLIBC` or Windows API) functions that work on network input. For example, suppose that the `GLIBC` function `fopen` builds its parameters upon an input message; we can infer that the part of the message related to file access modes (such as '`read`' and '`write`') can be less sensitive than that concerning file name. This approach needs a model of API functions and trust in the information provided by the server. Another solution is to partition an input stream using a set of tokens and common delimiters such as '`\n`'. Such tokens can be specified by the user. For example, using the token '`secret`' and the delimiter '`.`', we can divide the URL '`www.secretservice.gov`' into the following fields: '`www`', '`.`', '`secretservice`' and '`gov`'. Upon these fields, different leakage thresholds can be defined. These two approaches can work together and also be applied to specify finer-grained policies within a protocol field when the protocol is public.

To facilitate specification of the privacy policies, Panalyst can provide the user with policy templates set by the expert. Such an expert can be any party who has the knowledge about fields and the amount of information that can be disclosed without endangering the content of a field. For example, people knowledgeable about the HTTP specifications are in the position to label the fields like '`www`' as nonsensive and domain names such as '`secretservice.gov`' as sensitive. Typically, protocol keywords, delimiters and some API parameters can be treated as public information, while the fields such as those including the tokens and other API parameters are deemed sensitive. A default leakage threshold for a sensitive field can be just a few bytes: for example, we can allow one or two bytes to be disclosed from a domain-name field, because they are too general to be used to pinpoint the domain name; as another example, up to four bytes can be exposed from a field that may involve credit-card numbers, because people usually tolerate such information leaks in real life. Note that we may not be able to assign a zero threshold to a sensitive field because this can easily cause an analysis to fail: to proceed with an analysis, the server often needs to know whether the field contains some special byte such as a delimiter, which gives away a small amount of information regarding its content. These policy templates can be adjusted by a user to define her customized policies.

**Policy enforcement.** To enforce privacy policies, we need to quantify the information leaked by the client's answers. This is straightforward in some cases but less so in others. For example, we know that answering 'yes' to the question "$B_1B_2B_3B_4$ = 'POST'?" in Figure 2 gives away four bytes; however, information leaks can be more difficult to gauge when it comes to the questions like "$B_j \times B_k < 256?$ ", where $B_j$ and $B_k$ indicates the $j$th and the $k$th bytes on a message respectively. Without loss of generality, let us consider a set of bytes $(B_{k[1]}, \ldots, B_{k[m]})$ of a protocol message, whose concrete values on the message makes a condition "$C(B_{k[1]}, \ldots, B_{k[m]}) \leq 0$" true. To quantify the information an answer to the question gives away, we need to know $\rho$, the proportion of all possible values these bytes can take that make the condition true. Finding $\rho$ is nontrivial because the set of the values these bytes can have can be very large, which makes it impractical to check them one by one against the inequality. Our solution to the problem is based upon the classic statistic technique for estimating a proportion in a population. Specifically, we randomly pick up a set of values for these bytes to verify a branching condition and repeat the trial for $n$ times. From these $n$ trials, we can estimate the proportion $\rho$ as $\frac{x}{n}$ where $x$ is the number of trials in which the condition is true. The accuracy of this estimate is described by the probability that a range of values contain the true value of $\rho$. The range here is called *confidence interval* and the probability called *confidence level*. Given a confidence interval and a confidence level, standard statistic technique can be used to determine the size of samples $n$ [2]. For example, suppose the estimate of $\rho$ is 0.3 with a confidence interval $\pm 0.5$ and a confidence level 0.95, which intuitively means $0.25 < \rho < 0.35$ with a probability 0.95; in this case, the number of trials we need to play is 323. This approach offers an approximation of information leaks: in the prior example, we know that with 0.95 confidence, information being leaked will be no more than $-\log_2 0.25 = 4$ bits. Using such an estimate and a predetermined leakage threshold, a policy enforcer can decide whether to let the client answer a question.

## 3.3 Panalyst Server

Panalyst server starts working on a vulnerable application upon receiving an initial error report from the client. The report includes the type of the error, and other non-sensitive information such as the corrupted pointer, the lengths of individual packets' application-level payloads and the content of public fields. Based upon it, the server conducts an instruction-level analysis of the application's executable, which we elaborate as follows.

**Taint analysis and symbolic execution.** Panalyst server performs a dynamic taint analysis on the vulnerable program, using a network input built upon the initial report as a taint source. The input involves a set of packets, whose application-layer payloads form a message characterized by the same length as the client's message and the information disclosed by the report. The server monitors the execution of the program instruction by instruction to track tainted data according to a set of taint-propagation rules. These rules are similar to those used in other taint-analysis techniques such as RIFLE [51], TaintCheck [44] and LIFT [45], examples of which are presented in Table 1. Along with the dynamic analysis, the server also performs a symbolic execution [37] that statically evaluates the execution of the program through interpreting its instructions, using symbols instead of real values as input. Each symbol used by Panalyst represents one byte on the input message. Analyzing the program in this way, we can not only keep close track of tainted data flows, but also formulate a symbolic expression for every tainted value in memory and registers.

Whenever the execution encounters a conditional branching with its condition tainted by input symbols, the server sends the condition as a question to the client to seek answer. With the answer from the client, the server can find *hypothetic values* for these symbols using a constraint solver. For example, a "no" to the question $B_i =$ '$\backslash n$' may result in a letter 'a' to be assigned to the $i$th byte on the input. To keep the runtime data consistent with the hypothetic value of symbol $B_i$, the server updates all the tainted values related to $B_i$ by evaluating their symbolic expressions with the hypothetic value. It is important to note that $B_i$ may appear in multiple branching conditions ($C_1 \leq 0, \ldots, C_k \leq 0$). Without loss of generality, suppose all of them are true. To find a value for $B_i$, the constraint solver must solve the constraint $(C_1 \leq 0) \wedge \ldots \wedge (C_k \leq 0)$. The server also needs to "refresh" the tainted values concerning $B_i$ each time when a new hypothetic value of the symbol comes up.

The server also queries the client when the program attempts to access memory through a pointer tainted by input symbols $(B_{k[1]}, \ldots, B_{k[m]})$. In this case, the server needs to give the symbolic expression of the pointer $S(B_{k[1]}, \ldots, B_{k[m]})$ to the client to get its value $v$, and solve the constraint $S(B_{k[1]}, \ldots, B_{k[m]}) = v$ to find these symbols' hypothetic values. Query of a tainted pointer is necessary for ensuring the program's correct execution, particularly when a write happens through such a pointer. It is also an important step for reliably reproducing a runtime error, as the server may need to know the value of a pointer, or at least its range, to determine whether an illegal memory access is about to occur. However, this treatment may disclose too much user information, in particular when the pointer involves only one symbol: a "yes" to such a question often exposes the real value of that symbol. Such a problem usually happens in a string-related `GLIBC` function, where letters on a string are used as offsets to look up a table. Our solution is to accommodate symbolic pointers in our analysis if such a pointer carries only one symbol and is used to read from a memory location. This approach can be explicated through an example. Consider the instruction "`MOV EAX, [ESI+CL]`", where `CL` is tainted by an input byte $B_j$. Instead of directly asking the client for the value of `ESI+CL`, which reveals the real value of $B_j$, the server gathers the bytes from the memory locations pointed by (`ESI+0`, `ESI+1`, ..., `ESI+ 255`) to form a list. The list is used to prepare a question should `EAX` get involved in a branching condition such as "`CMP EAX, 1`". In this case, the server generates a query including `[ESI+CL]`, which is the symbolic expression of `EAX`, the value of `ESI`, the list and the condition. In response to the query, the client uses the real value of $B_j$ and the list to verify the condition and answer either "yes" or "no", which enables the server to identify the right branch.

Table 1: Examples of the Taint Rules.

| Instruction Category | Taint Propagation | Examples |
|---|---|---|
| data movement | (1) taint is propagated to the destination if the source is tainted, (2) the destination operand is not tainted if the source operand is not tainted. | `mov eax,ebx;` `push eax;` `call 0x4080022;` `lea ebx, ptr [ecx+10]` |
| arithmetic | (1) taint is propagated to the destination if the source is tainted, (2) the EFLAGS is also regarded as a destination operand. | `and eax, ebx;` `inc ecx;` `shr eax,0x8` |
| address calculation | an address is tainted if any element in the address calculation is tainted | `mov ebx, dword ptr [ecx+2*ebx+0x08]` |
| conditional jump | regard EFLAGS as a source operand | `jz 0x0746323;` `jnle 0x878342;` `jg 0x405687` |
| compare | regard EFLAGS as a destination operand | `cmp eax,ebx;` `test eax,eax` |

The analysis stops when the execution reaches a state where a runtime error is about to happen. Examples of such a state include a jump to an address outside the process image or an illegal instruction, and memory access through an illegal pointer. When this happens, Panalyst server announces that an input reproducing the error has been identified, and can be used for further analysis of the underlying bug and generation of signatures [52, 50, 39] or patches [49]. Our analysis also contributes to a preliminary classification of bugs: if the illegal address that causes the error is found to be tainted, we have a reason to believe that the underlying bug can be exploited remotely and therefore is security critical.

**Reducing communication overhead.** A major concern for Panalyst seems to be communication overhead: the server may need to query the client whenever a tainted branching condition or a tainted pointer is encountered. However, in our research, we found that the bandwidth consumed in an analysis usually is quite small, less than a hundred KB during the whole analysis. This is because the number of tainted conditions and pointers can be relatively small in many programs, and both the server's questions and the client's answers are usually short. Need for communication can be further reduced if an initial error report supplies the server with a sufficient amount of public information regarding the error. However, the performance of the server and the client will still be affected when the program intensively operates on tainted data, which in many cases is related to loop.

A typical loop that appears in many network-facing applications is similar to the one in the example (Line 6 of Figure 2). The loop compares individual bytes in a protocol field with a delimiter such as '$\backslash n$' or ' ' to identify the end of the field. If we simply view the loop as a sequence of conditional branching, then the server has to query the client for every byte within that field, which can be time consuming. To mitigate this problem, we designed a technique in our research to first identify such a loop and then let client proactively scan its message to find the location of the first string that terminates the loop. We describe the technique below.

The server monitors a tainted conditional branching that the execution has repeatedly bumped into. When the number of such encounters exceeds a threshold, we believe that a loop has been identified. The step value of that loop can be approximated by the difference between the indices of the symbols that appear in two consecutive evaluations of the condition. For example, consider the loop in Figure 2. If the first time the execution compares $B_j$ with '$\backslash n$' and the second time it tries $B_{j+1}$, we estimate the step as one. The server then sends a question to the client, including the loop condition $C(B_{k[1]}, \ldots, B_{k[m]})$ and step estimates $\lambda_{k[1]}, \ldots, \lambda_{k[m]}$. The client starts from the $k[i]$th byte ($1 \leq i \leq m$) to scan its message every $\lambda_{k[j]}$ bytes, until it finds a set of bytes $(B'_{k[1]}, \ldots, B'_{k[m]})$ that makes the condition false. The positions of these bytes are shipped to the server. As a result, the analysis can evaluate the loop condition using such information, without talking to the client iteration by iteration.

The above technique only works on a simple loop characterized by a constant step value. Since such a loop frequently appears in network-facing applications, our approach contributes to significant reduction of communication when analyzing these applications. Development of a more general approach for dealing with the loops with varying step size is left as our future research. Another problem of our technique is that the condition it identifies may not be a real loop condition. However, this does not bring us much trouble in general, as the penalty of such a false positive can be small, including nothing but the requirement for the client to scan its message and disclosure of a few bytes that seem to meet the exit condition. If the client refuses to do so, the analysis can still continue through directly querying the client about branching conditions.

**Improving constraint-solving performance.** Solving a constraint can be time consuming, particularly when the constraint is nonlinear, involving operations such as bitwise AND, OR and XOR. To maintain a valid runtime state for the program under analysis, Panalyst server needs to run a constraint solver to update hypothetic symbol values whenever a new branching condition or memory access is encountered. This will impact the server's performance. In our research, we adopted a very simple strategy to mitigate this impact: we check whether current hypothetic values satisfy a new constraint before solving the constraint. This turns out to be very effective: in many cases, we found that symbol values good for an old constraint also work for a new constraint, which allows us to skip the constraint-solving step.

## 4 Implementation

We implemented a prototype of Panalyst under Linux, including its server component and client component. The details of our implementation are described in this section.

**Message logging.** We adopted `ptrace` to dump the packet payloads an application receives. Specifically, `ptrace` intercepts the system call `socketcall()` and parses its parameters to identify the location of an input buffer. The content of the buffer is dumped to a log file. We also labels the beginning of a connection when an `accept()` is observed and the end of the connection when there is a `close()`. The data between these two calls are used to build a message once a runtime exception happens to the application.

**Estimate of information leaks.** To evaluate the information leaks caused by answering a question, our implementation first generates a constraint that is a conjunction of all the constraints the client receives that are directly or transitively related to the question, and then samples values of the constraint using the random values of the symbols it contains. We set the number of samples to 400, which achieves a confidence interval of $\pm 0.05$ and a confidence level of 0.95. A problem here is that the granularity of the control here could be coarse, as 400 samples can only represent loss of one byte of information. When this happens, our current implementation takes a conservative treatment to assume that all the bytes in a constraint are revealed. A finer-grained approach can be restoring the values of the symbols byte by byte to repeatedly check information leaks, until all the bytes are disclosed. An evaluation of such an approach is left as our future work.

**Error analyzer.** We implemented an error analyzer as a Pin tool that works under Pin's Just-In-Time (JIT) mode [40]. The analyzer performs both taint analysis and symbolic execution on a vulnerable application, and builds a new input to reproduce the runtime error that occurred on the client. The analyzer starts from a message that contains nothing but zeros and has the same length as the client's input, and designates a symbol to every byte on that message. During the analysis, the analyzer first checks whether a taint will be propagated by an instruction and only symbolically evaluates those whose operands involve tainted bytes. Since many instructions related to taint propagation use the information of `EFLAGS`, the analyzer also takes this register as a source operand for these instructions. Once an instruction's source operand is tainted, symbolic expressions are computed for the destination operand(s). For example, consider the instruction `add eax, ebx`, where `ebx` is tainted. Our analyzer first computes a symbolic expression $B_{ebx} + v_{eax}$, where $B_{ebx}$ is an expression for `ebx` and $v_{eax}$ is the value of `eax`, and then generates another expression for `EFLAGS` because the result of the operation affects Flag `OF, SF, ZF, AF, CF, PF`.

Whenever a conditional jump is encountered, the server queries the client about `EFLAGS`. To avoid asking the client to give away too much information, such a query only concerns the specific flag that affects that branching, instead of the whole status of `EFLAGS`. For example, consider the following branching: `cmp eax,ebx` and then `jz 0x33fd740`. In this case, the server's question is only limited to the status of `ZF`, which the branching condition depends on, though the comparison instruction also changes other flags such as `SF` and `CF`.

**Constraint solver.** Our implementation uses Yices [33] to solve constraints so as to find the hypothetic values for individual symbols. These values are important to keeping the application in a state that is consistent with its input. Yices is a powerful constraint solver which can handle many nonlinear constraints. However, there are situations when a constraint is so complicated that its solution cannot be obtained within a reasonable time. When this happens, we adopted a strategy that gradually inquires the client about the values of individual symbols to simplify the constraint, until it becomes solvable by the constraint solver.

**Data compression.** We implemented two measures to reduce the communication between the client and the server. The first one is for processing the questions that include the same constraints except input symbols. Our implementation indexes each question the server sends to the client. Whenever the server is about to ask a question that differs from a previous one only in symbols, it only transmits the index of the old question and these symbols. This strategy is found to be extremely effective when the sizes of the questions become large: in our experiment, a question with 8KB was compressed to 52 bytes. The strategy also complements our technique for processing loops: for a complicated loop with varying steps which the technique cannot handle, the server needs to query the client iteratively; however, the sizes of these queries can be very small as they are all about the same constraint with different symbols. The second measure is to use a lightweight real-time compression algorithm to reduce packet sizes. The algorithm we adopted is minilzo [6], which reduced the bandwidth consumption in our experiments to less than 100 KB for an analysis, at a negligible computational overhead.

## 5 Evaluation

In this section, we describe our experimental study of Panalyst. The objective of this study is to understand the effectiveness of our technique in remote error analysis and protection of the user's privacy, and the overheads it introduces. To this end, we evaluated our prototype using 6 real applications and report the outcomes of these experiments here.

Our experiments were carried out on two Linux workstations, one as the server and the other as the client. Both of them were installed with Redhat Enterprise 4. The server has a 2.40GHz Core 2 Duo processor and 3GB memory. The client has a Pentium 4 1.3GHz processor and 256MB memory.

### 5.1 Effectiveness

We ran Panalyst to analyze the errors that occurred in 6 real applications, including Newspost [7], OpenVMPS [19], Null-HTTPd (Nullhttpd) [8], Sumus [15], Light HTTPd [5] and ATP-HTTPd [3]. The experimental results are presented in Table 2. These applications contain bugs that are subject to stack-based overflow, format string error and heap-based overflow. The errors were triggered by a single or multiple input packets on the client and analyzed on the server. As a result, new packets were gradually built from an initial error report and interactions with the client to reproduce an error. This was achieved without leaking too much user information. We elaborate our experiments below.

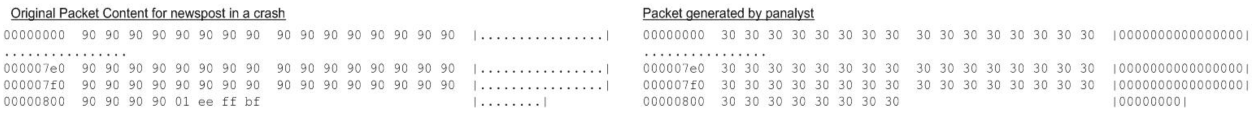
**Newspost.** Newspost is a Usenet binary autoposter for Unix and Linux. Its version 2.1.1 and earlier has a bug subject to stack-based overflow: specifically, a buffer in the `socket_getline()` function can be overrun by a long string without a newline character. In our experiment, the application was crashed by a packet of 2KB. After this happened, the client sent the server an initial error report that described the length of the packet and the type of the error. The report was converted into an input to an analysis performed on the application, which included an all-zero string of 2KB. During the analysis, the server identified a loop that iteratively searched for '`0xa`', the newline symbol, as a termination condition for moving bytes into a buffer, and questioned the client about the position at which the byte first appeared. The byte actually did not exist in the client's packet. As a result, the input string overflowed the buffer and was spilled on an illegal address to cause a segmentation fault. Therefore, the server's input was shown to be able to reproduce the error. This analysis was also found to disclose very little user information: nothing more than the fact that none of the input bytes were '`0xa`' were revealed. This was quantified as 0.9 byte.

**OpenVMPS.** OpenVMPS is an open-source implementation of Cisco Virtual Membership Policy Server, which dynamically assigns ports to virtual networks according to Ethernet addresses. The application has a format string bug which allows the input to supply a string with format specifiers as a parameter for `vfprintf()`. This could make `vfprintf()` write to a memory location. In the experiment, Panalyst server queried the client to get "`00 00 0c 02`" as illustrated in Figure 4. These four bytes were part of a branching condition, and seems to be a keyword of the protocol. We also found that the string "`00 b9`" were used as a loop counter. These two bytes were identified by the constraint solver. The string "`62637`" turned out to be the content that the format specifier "`%19$hn`" wrote to a memory location through `vfprintf()`. They were recovered from the client because they were used as part of a pointer to access memory. Our implementation successfully built a new input on the server that reproduced the error, as illustrated in Figure 4. This analysis recovered 39 bytes from the client, all of which were either related to branching conditions or memory access. An additional 18.4 bytes of information were estimated by the client to be leaked, as a result of the client's answers which reduced the ranges of the values some symbols could take.

**Null-HTTPd.** Null-HTTPd is a small web server working on Linux and Windows. Its version 0.5 contains a heap-overflow bug, which can be triggered when the HTTP request is a `POST` with a negative `Content Length` field and a long request content. In our experiment, the client parsed the request using Wireshark and delivered nonsensitive information such as the keyword `POST` to the server. The server found that the application added 1024 to the value derived from the `Content Length` and used the sum as pointer in the function `calloc`. This resulted in a query for the value of that field, which the client released. At this point, the server acquired all the information necessary for reproducing the error and generated a new input illustrated in Figure 5. The information leaks caused by the analysis include the keyword, the value of `Content Length`, HTTP delimiters and the knowledge that some bytes are not special symbols such as delimiters. This was quan-

Table 2: Effectiveness of Panalyst.

| Applications | Vul. Type | New Input Generated? | Size of client's message (bytes) | Info leaks (bytes) | Rate of info leaks |
|---|---|---|---|---|---|
| Newspost | Stack Overflow | Yes | 2056 | 0.9 | 0.04% |
| OpenVMPS | Format String | Yes | 199 | 57.4 | 28.8% |
| Null-HTTPd | Heap Overflow | Yes | 416 | 29.7 | 7.14% |
| Sumus | Stack Overflow | Yes | 500 | 7.7 | 1.54% |
| Light HTTPd | Stack Overflow | Yes | 211 | 17.9 | 8.48% |
| ATP-HTTPd | Stack Overflow | Yes | 819 | 16.7 | 2.04% |


Original Packet Content for newspost in a crash

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
................
000007e0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
000007f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000800  90 90 90 90 01 ee ff bf                           |........|
```

Packet generated by panalyst

```
00000000  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
000007e0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000007f0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000800  30 30 30 30 30 30 30 30                           |00000000|
```

Figure 3: Input Generation for Newspost. Left: the client's packet; Right: the new packet generated on the server.

tified as 29.7 bytes, about 7% of the HTTP message the client received.

**Sumus.** Sumus is a server for playing Spanish "mus" game on the Internet. It is known that Sumus 0.2.2 and the earlier versions have a vulnerable buffer that can be overflowed remotely [14]. In our experiment, Panalyst server gradually constructed a new input through interactions with the client until the application was found to jump to a tainted address. At this point, the input was shown to be able to reproduce the client's error. The information leaked during the analysis is presented in Figure 6, including a string "GET" which affected a path condition, and 4 "`0x90`", which were the address the application attempted to access. These 7 bytes were counted as leaked information, along with the fact that other bytes were not a delimiter.

**Light-HTTPd.** Light-HTTPd is a free HTTP server. Its version 0.1 has a vulnerable buffer on the stack. Our experiment captured an exception that happened when the application returned from the function `vsprintf()` and constructed the new input. The input shared 14 bytes with the client's input which were essential to determining branching conditions and accessing memory. For example, the keyword "`GET`" appeared on a conditional jump and the letter "`H`" were used as a condition in the `GLIBC` function `strstr`. The remaining 3.9 bytes were caused by the intensive string operations, such as `strtok`, which frequently used individual bytes for table lookup and comparison operations. Though these operations did not give away the real values of these bytes, they reduced the range of the bytes, which were quantified into another 3.9 bytes.

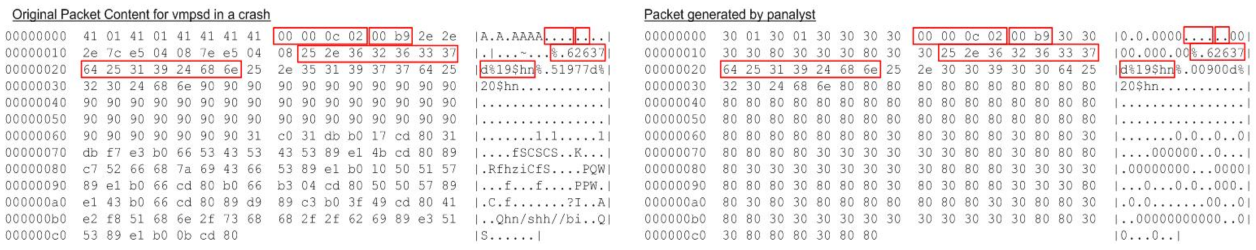
**ATP-HTTPd.** ATP-HTTPd 0.4 and 0.4b involve a remotely exploitable buffer in the `socket_gets()` function. A new input that triggered this bug was built in our experiment, which are presented in Figure 8. For example, the string "`EDCB`" was an address the application attempted to jump to; this operation actually caused a segmentation fault. Information leaks during this analysis are similar to that of Light-HTTPd, which was quantified as 16.7 bytes.

## 5.2 Performance

We also evaluated the performance of Panalyst. The client was deliberately run on a computer with 1 GHz CPU and 256MB memory to understand the performance impact of our technique on a low-end system. The server was on a high-end, with a 2.40GHz Core 2 Duo CPU and 3GB memory. In our experiments, we measured the delay caused by an analysis, memory use and bandwidth consumption on both the client and the server. The results are presented in Table 3.

The client's delay describes the accumulated time that the client spent to receive packets from the server, compute answers, evaluate information leaks and deliver the responses. In our experiments, we observed that this whole process incurred the latency below 3.2 seconds. Moreover, the memory use on the client side was kept below 5 MB. Given the hardware platform over which this performance was achieved, we have a reason to believe that such overhead could be afforded by even a device with limited computing resources, such as Pocket PC and PDA. Our analysis introduced a maximal 99,659 bytes communication overhead. We believe this is still reasonable for the client, because the size of a typical web page exceeds 100 KB and many mobile devices nowadays have the capability of web browsing.

The delay on the server side was measured between the reception of an initial error report and the generation of a new input. An additional 15 seconds for launching our Pin-based analyzer should also be counted. Given this, the server's performance was very good: the maximal latency was found to be under 1 minute. However,



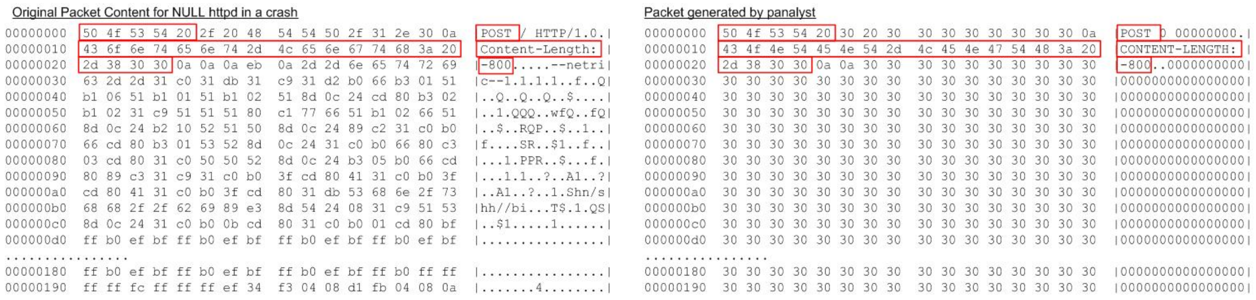
Original Packet Content for vmpsd in a crash

```
00000000  41 01 41 01 41 41 41 41  00 00 0c 02 00 b9 2e 2e  |A.A.AAAA........|
00000010  2e 7c e5 04 08 7e e5 04  08 25 2e 36 32 36 33 37  |.|...~...%.62637|
00000020  64 25 31 39 24 68 6e 25  2e 35 31 39 37 37 64 25  |d%19$hn%.51977d%|
00000030  32 30 24 68 6e 90 90 90  90 90 90 90 90 90 90 90  |20$hn...........|
00000040  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000050  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000060  90 90 90 90 90 90 90 31  c0 31 db b0 17 cd 80 31  |.......1.1.....1|
00000070  db f7 e3 b0 66 53 43 53  43 53 89 e1 4b cd 80 89  |....fSCSCS..K...|
00000080  c7 52 66 68 7a 69 43 66  53 89 e1 b0 10 50 51 57  |.RfhziCfS....PQW|
00000090  89 e1 b0 66 cd 80 b0 66  b3 04 cd 80 50 50 57 89  |...f...f....PPW.|
000000a0  e1 43 b0 66 cd 80 89 d9  89 c3 b0 3f 49 cd 80 41  |.C.f.......?I..A|
000000b0  e2 f8 51 68 6e 2f 73 68  68 2f 2f 62 69 89 e3 51  |..Qhn/shh//bi..Q|
000000c0  53 89 e1 b0 0b cd 80                              |S......|
```

Packet generated by panalyst

```
00000000  30 01 30 01 30 30 30 30  00 00 0c 02 00 b9 30 30  |0.0.0000......00|
00000010  30 30 80 30 30 30 80 30  30 25 2e 36 32 36 33 37  |00.000.00%.62637|
00000020  64 25 31 39 24 68 6e 25  2e 30 30 39 30 30 64 25  |d%19$hn%.00900d%|
00000030  32 30 24 68 6e 80 80 80  80 80 80 80 80 80 80 80  |20$hn...........|
00000040  80 80 80 80 80 80 80 80  80 80 80 80 80 80 80 80  |................|
00000050  80 80 80 80 80 80 80 80  80 80 80 80 80 80 80 80  |................|
00000060  80 80 80 80 80 80 80 30  80 30 80 80 30 80 80 30  |.......0.0..0..0|
00000070  80 80 80 80 30 30 30 30  30 30 80 80 30 80 80 80  |....000000..0...|
00000080  80 30 30 30 30 30 30 30  30 80 80 80 30 30 30 30  |.00000000...0000|
00000090  80 80 80 30 80 80 80 30  80 30 80 80 30 30 30 80  |...0...0.0..000.|
000000a0  80 30 80 30 80 80 80 80  80 80 80 30 30 80 80 30  |.0.0.......00..0|
000000b0  80 80 30 30 30 30 30 30  30 30 30 30 30 80 80 30  |..00000000000..0|
000000c0  30 80 80 80 30 80 80                              |0...0..|
```

Figure 4: Input Generation for OpenVMPS. Left: the client's packet; Right: the new packet generated on the server.



Original Packet Content for NULL httpd in a crash

```
00000000  50 4f 53 54 20 2f 20 48  54 54 50 2f 31 2e 30 0a  POST / HTTP/1.0.|
00000010  43 6f 6e 74 65 6e 74 2d  4c 65 6e 67 74 68 3a 20  Content-Length: |
00000020  2d 38 30 30 0a 0a 0a eb  0a 2d 2d 6e 65 74 72 69  |-800.....--netri|
00000030  63 2d 2d 31 c0 31 db 31  c9 31 d2 b0 66 b3 01 51  |c--1.1.1.1..f..Q|
00000040  b1 06 51 b1 01 51 b1 02  51 8d 0c 24 cd 80 b3 02  |..Q..Q..Q..$....|
00000050  b1 02 31 c9 51 51 51 80  c1 77 66 51 b1 02 66 51  |..1.QQQ..wfQ..fQ|
00000060  8d 0c 24 b2 10 52 51 50  8d 0c 24 89 c2 31 c0 b0  |..$..RQP..$..1..|
00000070  66 cd 80 b3 01 53 52 8d  0c 24 31 c0 b0 66 80 c3  |f....SR..$1..f..|
00000080  03 cd 80 31 c0 50 50 52  8d 0c 24 b3 05 b0 66 cd  |...1.PPR..$...f.|
00000090  80 89 c3 31 c9 31 c0 b0  3f cd 80 41 31 c0 b0 3f  |...1.1..?..A1..?|
000000a0  cd 80 41 31 c0 b0 3f cd  80 31 db 53 68 6e 2f 73  |..A1..?..1.Shn/s|
000000b0  68 68 2f 2f 62 69 89 e3  8d 54 24 08 31 c9 51 53  |hh//bi...T$.1.QS|
000000c0  8d 0c 24 31 c0 b0 0b cd  80 31 c0 b0 01 cd 80 bf  |..$1.....1......|
000000d0  ff b0 ef bf ff b0 ef bf  ff b0 ef bf ff b0 ef bf  |................|
................
00000180  ff b0 ef bf ff b0 ef bf  ff b0 ef bf ff b0 ff ff  |................|
00000190  ff ff fc ff ff ff ef 34  f3 04 08 d1 fb 04 08 0a  |.......4........|
```

Packet generated by panalyst

```
00000000  50 4f 53 54 20 30 20 30  30 30 30 30 30 30 30 0a  POST 0 00000000.|
00000010  43 4f 4e 54 45 4e 54 2d  4c 45 4e 47 54 48 3a 20  CONTENT-LENGTH: |
00000020  2d 38 30 30 0a 0a 30 30  30 30 30 30 30 30 30 30  -800..0000000000|
00000030  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000040  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000050  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000060  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000070  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000080  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000090  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000d0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
00000180  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000190  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
```

Figure 5: Input Generation for Null-HTTPd. Left: the client's packet; Right: the new packet generated on the server.

this was achieved on a very high-end system. Actually, we observed that the latency was doubled when moving the server to a computer with 2.36 GHz CPU and 1 GB memory. More importantly, the server consumed about 100 MB memory during the analysis. This can be easily afforded by a high-end system as the one used in our experiment, but could be a significant burden to a low-end system such as a mobile device. As an example, most PDAs have less than 100 MB memory. Therefore, we believe that Panalyst server should be kept on a dedicated high-performance system.

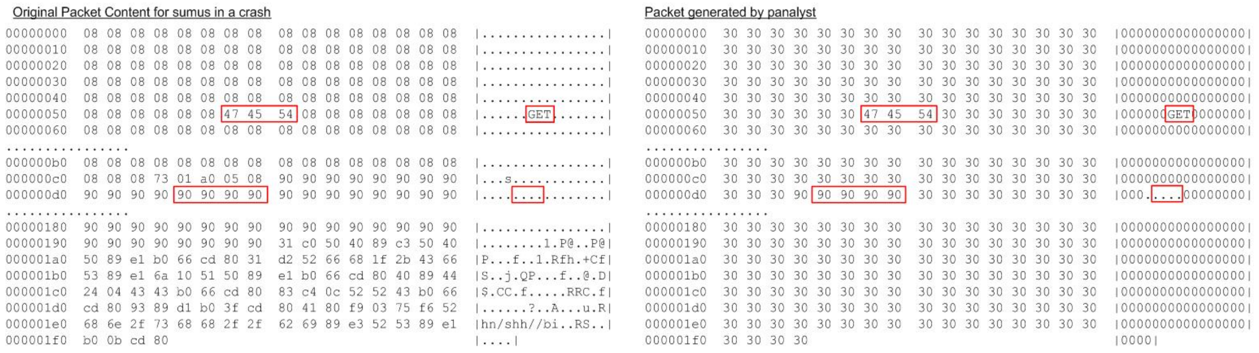
## 6 Discussion

Our research makes the first step towards a fully automated and privacy-aware remote error analysis. However, the current design of Panalyst is still preliminary, leaving much to be desired. For example, the approach does not work well in the presence of probabilistic errors, and our privacy policies can also be better designed. We elaborate limitations and possible solutions in the left part of this section, and discuss the future research for improving our technique in Section 7.

The current design of Panalyst is for analyzing the error triggered by network input alone. However, runtime errors can be caused by other inputs such as those from a local file or another process. Some of these errors can also be handled by Panalyst. For example, we can record all the data read by a vulnerable program and organize them into multiple messages, each of which corresponds to a particular input to the program; an error analysis can happen on these messages in a similar fashion as described in Section 3. A weakness of our technique is that it can be less effective in dealing with a probabilistic error such as the one caused by multithread interactions. However, it can still help the server build sanitized inputs that drive the vulnerable program down the same execution paths as those were followed on the client.

Panalyst may require the client to leak out some information that turns out to be unnecessary for reproducing an error, in particular, the values of some tainted pointer unrelated to the error. A general solution is describing memory addresses as symbolic expressions and taking them into consideration during symbolic execution. This approach, however, can be very expensive, especially when an execution involves a large amount of indirect addressing through the tainted pointers. To maintain a moderate overhead during an analysis, our current design only offers a limited support for symbolic pointers: we introduce such a pointer only when it includes a single symbol and is used for reading from memory.

The way we treat loops is still preliminary: it only works on the loops with constant step sizes and may falsely classify a branching condition as a loop condition. As a result, we may miss some real loops, which increases the communication overhead of an analysis, or require the client to unnecessarily disclose extra informa-



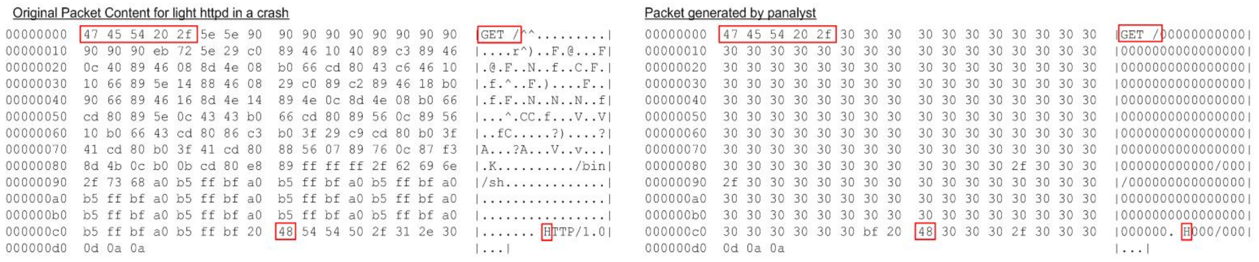
Original Packet Content for sumus in a crash

```
00000000  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000010  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000020  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000030  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000040  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
00000050  08 08 08 08 08 08 47 45  54 08 08 08 08 08 08 08  |......GET.......|
00000060  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
................
000000b0  08 08 08 08 08 08 08 08  08 08 08 08 08 08 08 08  |................|
000000c0  08 08 08 73 01 a0 05 08  90 90 90 90 90 90 90 90  |...s............|
000000d0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
................
00000180  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000190  90 90 90 90 90 90 90 90  31 c0 50 40 89 c3 50 40  |........1.P@..P@|
000001a0  50 89 e1 b0 66 cd 80 31  d2 52 66 68 1f 2b 43 66  |P...f..1.Rfh.+Cf|
000001b0  53 89 e1 6a 10 51 50 89  e1 b0 66 cd 80 40 89 44  |S..j.QP...f..@.D|
000001c0  24 04 43 43 b0 66 cd 80  83 c4 0c 52 52 43 b0 66  |$.CC.f.....RRC.f|
000001d0  cd 80 93 89 d1 b0 3f cd  80 41 80 f9 03 75 f6 52  |......?..A...u.R|
000001e0  68 6e 2f 73 68 68 2f 2f  62 69 89 e3 52 53 89 e1  |hn/shh//bi..RS..|
000001f0  b0 0b cd 80                                       |....|
```

Packet generated by panalyst

```
00000000  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000020  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000030  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000040  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000050  30 30 30 30 30 30 47 45  54 30 30 30 30 30 30 30  |000000GET0000000|
00000060  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
000000b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000d0  30 30 30 90 90 90 90 90  30 30 30 30 30 30 30 30  |000.....00000000|
................
00000180  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000190  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001d0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001e0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001f0  30 30 30 30                                       |0000|
```

Figure 6: Input Generation for Sumus. Left: the client's packet; Right: the new packet generated on the server.



Original Packet Content for light httpd in a crash

```
00000000  47 45 54 20 2f 5e 5e 90  90 90 90 90 90 90 90 90  |GET /^^.........|
00000010  90 90 90 eb 72 5e 29 c0  89 46 10 40 89 c3 89 46  |....r^)..F.@...F|
00000020  0c 40 89 46 08 8d 4e 08  b0 66 cd 80 43 c6 46 10  |.@.F..N..f..C.F.|
00000030  10 66 89 5e 14 88 46 08  29 c0 89 c2 89 46 18 b0  |.f.^..F.)....F..|
00000040  90 66 89 46 16 8d 4e 14  89 4e 0c 8d 4e 08 b0 66  |.f.F..N..N..N..f|
00000050  cd 80 89 5e 0c 43 43 b0  66 cd 80 89 56 0c 89 56  |...^.CC.f...V..V|
00000060  10 b0 66 43 cd 80 86 c3  b0 3f 29 c9 cd 80 b0 3f  |..fC.....?)....?|
00000070  41 cd 80 b0 3f 41 cd 80  88 56 07 89 76 0c 87 f3  |A...?A...V..v...|
00000080  8d 4b 0c b0 0b cd 80 e8  89 ff ff ff 2f 62 69 6e  |.K........../bin|
00000090  2f 73 68 a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |/sh.............|
000000a0  b5 ff bf a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |................|
000000b0  b5 ff bf a0 b5 ff bf a0  b5 ff bf a0 b5 ff bf a0  |................|
000000c0  b5 ff bf a0 b5 ff bf 20  48 54 54 50 2f 31 2e 30  |....... HTTP/1.0|
000000d0  0d 0a 0a                                          |...|
```

Packet generated by panalyst

```
00000000  47 45 54 20 2f 30 30 30  30 30 30 30 30 30 30 30  |GET /00000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000020  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000030  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000040  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000050  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000060  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000070  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000080  30 30 30 30 30 30 30 30  30 30 30 30 2f 30 30 30  |000000000000/000|
00000090  2f 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |/000000000000000|
000000a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000b0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000000c0  30 30 30 30 30 30 bf 20  48 30 30 30 2f 30 30 30  |000000. H000/000|
000000d0  0d 0a 0a                                          |...|
```

Figure 7: Input Generation for Light HTTPd. Left: the client's packet; Right: the new packet generated on the server.

tion. However, the client can always refuse to give more information and set a threshold for the maximal number of the questions it will answer. Even if this causes the analysis to fail, the server can still acquire some information related to the error and use it to facilitate other error analysis techniques such as fuzz testing. We plan to study more general techniques for analyzing loops in our future research.

Entropy-based policies may not be sufficient for regulating information leaks. For example, complete disclosure of one byte in a field may have different privacy implications from leakage of the same amount of information distributed among several bytes in the field. In addition, specification of such policies does not seem to be intuitive, which may affect their usability. More effective privacy policies can be built upon other definitions of privacy such as *k*-Anonymity [46], *l*-Diversity [41] and *t*-Closeness [38]. These policies will be developed and evaluated in our future work.

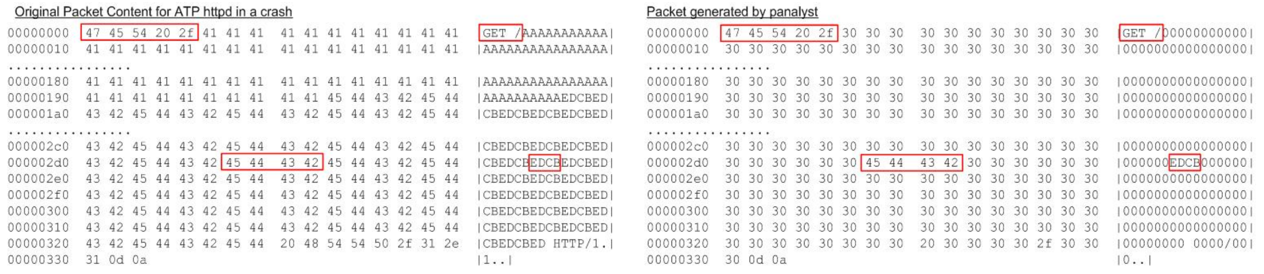
Panalyst client can only approximate the amount of information disclosed by its answers using statistical means. It also assumes a uniform distribution over the values a symbol can take. Design of a better alternative for quantifying and controlling information is left as our future research.

Another limitation of our approach is that it cannot handle encoded or encrypted input. This problem can be mitigated by interposing on the API functions (such as those in the OpenSSL library) for decoding or decryption to get their plaintext outputs. Our error analysis will be conducted over the plaintext.

## 7 Related Work

Error reporting techniques have been widely used for helping the user diagnose application runtime error. Windows error reporting [20], a technique built upon Microsoft's Dr. Watson service [18], generates an error report through summarizing a program state, including contents of registers and stack. It may also ask the user for extra information such as input documents to investigate an error. Such an error report is used to search an expert system for the solution provided by human experts. If the search fails, the client's error will be recorded for a future analysis. Crash Reporter [16] of `Mac OS X` and third-party tools such as BugToaster [27] and Bug Buddy [22] work in a similar way. As an example, Bug Buddy for GNOME can generate a stack trace using `gdb` and let the user post it to the GNOME bugzilla [4].

Privacy protection in existing error reporting techniques mostly relies on the privacy policies of those who collect reports. This requires the user to trust the collector, and also forces her to either send the whole report


Original Packet Content for ATP httpd in a crash

```
00000000  47 45 54 20 2f 41 41 41  41 41 41 41 41 41 41 41  GET /AAAAAAAAAAA|
00000010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
................
00000180  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000190  41 41 41 41 41 41 41 41  41 41 45 44 43 42 45 44  |AAAAAAAAAAEDCBED|
000001a0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
................
000002c0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002d0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002e0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
000002f0  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000300  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000310  43 42 45 44 43 42 45 44  43 42 45 44 43 42 45 44  |CBEDCBEDCBEDCBED|
00000320  43 42 45 44 43 42 45 44  20 48 54 54 50 2f 31 2e  |CBEDCBED HTTP/1.|
00000330  31 0d 0a                                          |1..|
```

Packet generated by panalyst

```
00000000  47 45 54 20 2f 30 30 30  30 30 30 30 30 30 30 30  |GET /00000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
00000180  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000190  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000001a0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
................
000002c0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000002d0  30 30 30 30 30 30 45 44  43 42 30 30 30 30 30 30  |000000EDCB000000|
000002e0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
000002f0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000300  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000310  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000320  30 30 30 30 30 30 30 30  20 30 30 30 30 2f 30 30  |00000000 0000/00|
00000330  30 0d 0a                                          |0..|
```

Figure 8: Input Generation for ATP HTTPd. Left: the client's packet; Right: the new packet generated on the server.

Table 3: Performance of Panalyst.

| Programs | client delay (s) | client memory use (MB) | server delay (s) | server memory use (MB) | total size of questions (bytes) | total size of answers (bytes) |
|---|---|---|---|---|---|---|
| Newspost | 0.022 | 4.7 | 12.14 | 99.3 | 527 | 184 |
| OPenVMPS | 1.638 | 3.9 | 17 | 122.3 | 45,610 | 6,088 |
| Null-HTTPd | 1.517 | 5.0 | 13.09 | 118.1 | 99,659 | 3,416 |
| Sumus | 0.123 | 4.8 | 1.10 | 85.4 | 5,968 | 2,760 |
| Light HTTPd | 0.88 | 4.8 | 6.59 | 110.1 | 14,005 | 2,808 |
| ATP-HTTPd | 3.197 | 5.0 | 37.11 | 145.4 | 50,615 | 15,960 |

or submit nothing at all. In contrast, Panalyst reduces the user's reliance on the collectors to protect her privacy and also allows her to submit part of the information she is comfortable with. Even if such information is insufficient for reproducing an error, it can make it easier for other techniques to identify the underlying bug. Moreover, Panalyst server can automatically analyze the error caused by an unknown bug, whereas existing techniques depend on human to analyze new bugs.

Proposals have been made to improve privacy protection during error reporting. Scrash [25] instruments an application's source code to record information related to a crash and generate a "clean" report that does not contain sensitive information. However, it needs source code and therefore does not work on commodity applications without the manufacturer's support. In addition, the technique introduces performance overheads even when the application works properly, and like other error reporting techniques, uses a remote expert system and therefore does not perform automatic analysis of new errors. Brickell, et al propose a privacy-preserving diagnostic scheme, which works on binary executables [24, 36]. The technique aims at searching a knowledge base framed as a decision tree in a privacy-preserving manner. It also needs to profile an application's execution. Panalyst differs from these approaches in that it does not interfere with an application's normal run except logging inputs, which is very lightweight, and is devised for automatically analyzing an unknown bug.

Techniques for automatic analysis of software vulnerabilities have been intensively studied. Examples include the approach for generating vulnerability-based signatures [26], Vigilante [30], DACODA [31] and EXE [53]. These approaches assume that an input triggering an error is already given and therefore privacy is no longer a concern. Panalyst addresses the important issue on how to get such an input without infringing too much on the user's privacy. This is achieved when Panalyst server is analyzing the vulnerable program. Our technique combines dynamic taint analysis with symbolic execution, which bears some similarity to a recent proposal for exploring multiple execution paths [42]. However, that technique is primarily designed for identifying hidden actions of malware, while Panalyst is for analyzing runtime errors. Therefore, we need to consider the issues that are not addressed by the prior approach. A prominent example is the techniques we propose to tackle a tainted pointer, which is essential to reliably reproducing an error.

Similar to Panalyst, a technique has been proposed recently to symbolically analyze a vulnerable executable and generate an error report through solving constraints [29]. The technique also applies entropy to quantify information loss caused by the error reporting. Panalyst differs from that approach fundamentally in that our technique generates a new input remotely while the prior approach directly works on the causal input on the client. Performing an intensive analysis on the client is exactly the thing we want to avoid, because this increases the client's burden and thus discourages the user from participating. Although an evaluation of the technique reports a moderate overhead [29], it does not include computation-intensive operations such as instruction-level tracing, which can, in some cases, introduce hundreds of seconds of delay and hundreds of megabytes of execution traces [23]. This can be barely acceptable to the user having such resources, and hardly affordable to those using weak devices such as PocketPC and PDA. Actually, reproducing an error without direct access to the causal input is much more difficult than analyzing the input locally, because it requires a careful coordination between the client and the server to ensure a gradual release of the input information without endangering the user's privacy and failing the analysis at the same time. In addition, Panalyst can enforce privacy policies to individual protocol fields and therefore achieves a finer-grained control of information than the prior approach.

## 8 Conclusion and Future Work

Remote error analysis is essential to timely discovery of security critical vulnerabilities in applications and generation of fixes. Such an analysis works most effectively when it protects users' privacy, incurs the least performance overheads on the client and provides the server with sufficient information for an effective study of the underlying bugs. To this end, we propose Panalyst, a new techniques for privacy-aware remote error analysis. Whenever a runtime error occurs, the Panalyst client sends the server an initial error report that includes nothing but the public information about the error. Using an input built from the report, Panalyst server analyzes the propagation of tainted data in the vulnerable application and symbolically evaluates its execution. During the analysis, the server queries the client whenever it does not have sufficient information to determine the execution path. The client responds to a question only when the answer does not leak out too much user information. The answer from the client allows the server to adjust the content of the input through symbolic execution and constraint solving. As a result, a new input will be built which includes the necessary information for reproducing the error on the client. Our experimental study of this technique demonstrates that it exposes a very small amount of user information, introduces negligible overheads to the client and enables the server to effectively analyze an error.

The current design of Panalyst is for analyzing the error triggered by network inputs alone. Future research will extend our approach to handle other types of errors. In addition, we also plan to improve the techniques for estimating information leaks and reduce the number of queries the client needs to answer.

## 9 Acknowledgements

We thank our Shepherd Anil Somayaji and anonymous reviewers for the comments on the paper. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

## References

[1] Wireshark. http://www.wireshark.org/.

[2] *NIST/SEMATECH e-Handbook of Statistical Methods*. http://www.itl.nist.gov/div898/handbook/, 2008.

[3] Athttpd Remote GET Request Buffer Overrun Vulnerability. http://www.securityfocus.com/bid/8709/discuss/, as of 2008.

[4] GNOME bug tracking system. http://bugzilla.gnome.org/, as of 2008.

[5] LIGHT http server and content management system. http://lhttpd.sourceforge.net/, as of 2008.

[6] miniLZO, a lightweight subset of the LZO library. http://www.oberhumer.com/opensource/lzo/#minilzo, as of 2008.

[7] Newspost, a usenet binary autoposter for unix. http://newspost.unixcab.org/, as of 2008.

[8] NullLogic, the Null HTTPD server. http://nullwebmail.sourceforge.net/httpd/, as of 2008.

[9] Privacy Statement for the Microsoft Error Reporting Service. http://oca.microsoft.com/en/dcp20.asp, as of 2008.

[10] Process Tracing Using Ptrace. http://linuxgazette.net/issue81/sandeep.html, as of 2008.

[11] Reducing Support Costs with Windows Vista. http://technet.microsoft.com/en-us/windowsvista/aa905076.aspx, as of 2008.

[12] Speed up Windows Mobile 5 pocket device . http://www.mobiletopsoft.com/board/388/speed-up-windows-mobile-5-pocket-device.html, as of 2008.

[13] Speed Up Windows Vista. http://www.extremetech.com/article2/0,1697,2110598,00.asp, as of 2008.

[14] Sumus Game Server Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/13162, as of 2008.

[15] SUMUS, the mus server. http://sumus.sourceforge.net/, as of 2008.

[16] Technical Note TN2123, CrashReporter. http://developer.apple.com/technotes/tn2004/tn2123.html, as of 2008.

[17] Tip: Disable Error reporting in Windows Mobile 5 to get better performance: msg#00043. http://osdir.com/ml/handhelds.ipaq.ipaqworld/2006-05/msg00043.html, as of 2008.

[18] U.S. Department of Energy Computer Incident Advisory Capability. Office XP Error Reporting May Send Sensitive Documents to Microsoft. http://www.ciac.org/ciac/bulletins/m-005.shtml, as of 2008.

[19] VMPS, VLAN Management Policy Server. http://vmps.sourceforge.net/, as of 2008.

[20] Windows Error Reporting. http://msdn2.microsoft.com/en-us/library/bb513641%28VS.85%29.aspx, as of 2008.

[21] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security* (2005), pp. 340–353.

[22] BERKMAN, J. Project Info for Bug-Buddy. http://www.advogato.org/proj/bug-buddy/, as of 2008.

[23] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (2006), pp. 154–163.

[24] BRICKELL, J., PORTER, D. E., SHMATIKOV, V., AND WITCHEL, E. Privacy-preserving remote diagnostics. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (2007), pp. 498–507.

[25] BROADWELL, P., HARREN, M., AND SASTRY, N. Scrash: A system for generating secure crash information. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 273–284.

[26] BRUMLEY, D., NEWSOME, J., SONG, D. X., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *S&P* (2006), pp. 2–16.

[27] BUGTOASTER. Do Something about computer Crashes. http://www.bugtoaster.com, 2002.

[28] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *OSDI* (2006), pp. 147–160.

[29] CASTRO, M., COSTA, M., AND MARTIN, J.-P. Better bug reporting with better privacy. In *Proceedings of Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)* (2008).

[30] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A. I. T., ZHOU, L., ZHANG, L., AND BARHAM, P. T. Vigilante: end-to-end containment of internet worms. In *Proceedings of SOSP* (2005), pp. 133–147.

[31] CRANDALL, J. R., SU, Z., AND WU, S. F. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 235–248.

[32] CUI, W., PAXSON, V., WEAVER, N., AND KATZ, R. H. Protocol-independent adaptive replay of application dialog. In *NDSS* (2006).

[33] DUTERTRE, B., AND MOURA, L. The YICES SMT Solver. http://yices.csl.sri.com/, as of 2008.

[34] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)* (June 2007).

[35] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *NDSS* (2008).

[36] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007), pp. 101–111.

[37] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[38] LI, N., LI, T., AND VENKATASUBRAMANIAN, S. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE* (2007), IEEE, pp. 106–115.

[39] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 213–222.

[40] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 190–200.

[41] MACHANAVAJJHALA, A., KIFER, D., GEHRKE, J., AND VENKITASUBRAMANIAM, M. L-diversity: Privacy beyond k-anonymity. *TKDD 1*, 1 (2007).

[42] MOSER, A., KRÜGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy* (2007), pp. 231–245.

[43] NEWSOME, J., BRUMLEY, D., FRANKLIN, J., AND SONG, D. X. Replayer: automatic protocol replay by binary analysis. In *ACM Conference on Computer and Communications Security* (2006), pp. 311–321.

[44] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[45] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO* (2006), pp. 135–148.

[46] SAMARATI, P., AND SWEENEY, L. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression, 1998. Technical Report SRI-CSL-98-04.

[47] SHANNON, C., AND MOORE, D. The spread of the witty worm. *IEEE Security & Privacy 2*, 4 (July/August 2004), 46–50.

[48] SHANNON, C. E. A mathematical theory of communication. *Bell system technical journal 27* (1948).

[49] SIDIROGLOU, S., AND KEROMYTIS, A. D. Countering network worms through automatic patch generation. *IEEE Security and Privacy 3*, 6 (2005), 41–49.

[50] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *Proceedings of OSDI* (2004), pp. 45–60.

[51] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO* (2004), IEEE Computer Society, pp. 243–254.

[52] WANG, X., LI, Z., XU, J., REITER, M. K., KIL, C., AND CHOI, J. Y. Packet vaccine: black-box exploit detection and signature generation. In *ACM Conference on Computer and Communications Security* (2006), pp. 37–46.

[53] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006), pp. 243–257.

# Multi-flow Attacks Against Network Flow Watermarking Schemes

Negar Kiyavash Amir Houmansadr Nikita Borisov
*Dept. of Computer Science* *Dept. of Electrical and Computer Engineering*
*University of Illinois at Urbana–Champaign*
*Email:* {`kiyavash,ahouman2,nikita`}`@uiuc.edu`

## Abstract

We analyze several recent schemes for watermarking network flows based on splitting the flow into intervals. We show that this approach creates time dependent correlations that enable an attack that combines multiple watermarked flows. Such an attack can easily be mounted in nearly all applications of network flow watermarking, both in anonymous communication and stepping stone detection. The attack can be used to detect the presence of a watermark, recover the secret parameters, and remove the watermark from a flow. The attack can be effective even if different the watermarks in different flows carry different messages.

We analyze the efficacy of our attack using a probabilistic model and a Markov-modulated Poisson process (MMPP) model of interactive traffic. We also implement our attack and test it using both synthetic and real-world traces, showing that our attack is effective with as few as 10 watermarked flows. Finally, we propose a countermeasure that defeats the attack by using multiple watermark positions.

## 1 Introduction

Traffic analysis is the practice of inferring sensitive information from communication patterns. Traffic analysis has been particularly studied in the context of anonymous communication systems, where features such as packet timings, sizes, and counts can be used to link two flows and break anonymity guarantees [2, 22]. Traffic analysis is also sometimes used in intrusion detection, for example, to detect the presence of stepping stones within an enterprise [29].

Recently, there has been a growing interest in the use of *watermarking* to aid traffic analysis [27, 24, 21, 25, 28]. In this case, traffic patterns of one flow (usually packet timings) are actively modified to contain a special pattern. If the same pattern is later found on another flow, the two are considered linked. Watermarking significantly reduces the computation and communication costs of traffic analysis, and may also lead to more precise detection with fewer false positives.[1] Watermarking has been applied to both the problems of attacking anonymity systems [24, 25, 28] and detecting stepping stones [27, 21].

In both contexts, many flows must be watermarked before linked flows are discovered. In our work, we consider whether an attacker can learn enough information to defeat the watermark by observing multiple watermarked flows. (We use "attacker" here to refer to someone attacking the watermarking scheme; in the case where watermarks themselves are used by attackers, these will be the "counter-attackers.") We apply this multi-flow threat model to the latest generation of *interval-based watermarks* [21, 25, 28]. These watermarks subdivide the flow to be marked into discrete time intervals and perform transformative operations on an entire interval of packets. This approach is more robust to packet losses, insertions, and repacketization than previous approaches that focused on individual packets [27, 24], because the time intervals allow the watermarker and detector to retain synchronization. However, the same synchronization property can be used by attackers by "lining up" multiple watermarked flows and observing the transformations that were inserted.

We show through experiments that the interval-based watermark schemes are completely vulnerable to an attacker who can collect a small number of watermarked flows—about 10. This is sufficient to not only detect that a watermark is indeed present, but also to recover the secret parameters of the watermark scheme and to be able to remove the watermark at a low cost. Furthermore, our attack works even if different watermarked flows contain different embedded "messages," with only about twice the number of watermarked flows necessary.

We also consider some countermeasures to such attacks. We show that by using multiple "keys" (time interval assignments) to watermark different flows, it is possible to defeat our attack. This countermeasure comes at a cost of higher computation overhead at the detector and a higher rate of false positives. However, this increased cost is only linear, whereas the increased cost for the attacker is superexponential, thus providing an effective defense.

The rest of the paper is organized as follows. The next section presents the setting for our attack and reviews the three schemes considered in this paper. Section 3 describes the theoretical foundation for our attack, and Section 4 implements the attack. We discuss potential countermeasures to the attack in Section 5. Section 6 concludes.

## 2 Background

We first describe the setting of our attack in a bit more detail and then review the essential details of the watermarking schemes we analyze.

### 2.1 Network Flow Watermarking

The setting for network flow watermarking is similar to that of other digital media watermarks (and network flow watermarks use similar techniques). The general model, as shown in Figure 1, involves a network flow passing through a watermarking point (typically a router of some sort) that transforms, or *distorts*, the flow in some way (typically by modifying packet timings by selectively delaying some packets). In the general setting, the watermarker has a secret *key* and uses it to encode a *message* in the traffic characteristics.

After watermarking, the flow undergoes some natural or intentional distortion. Natural distortion can take the form of delays at intermediate routers (or rather, variability of delays, i.e., *jitter*), but may also include dropped or retransmitted packets, repacketization, and other changes. In addition, an attacker may intentionally distort traffic characteristics in order to prevent the watermark from being recovered.

The distorted flow finally arrives at a detection point. The detector shares the secret key and uses it to extract the message encoded in the watermark. A good watermark will allow reliable recovery of the message from the watermarked flow despite the intermediate distortion.

In network flow watermarks, the *message* component of the watermark may be used in two ways. First, all watermarked flows may be marked with a single message. In this case, the detector's main goal is to decide whether the watermark is present or not by checking whether the decoded message is the correct one. Alternately, different flows may have a different message embedded, so that when a watermarked flow is detected, it can be linked with a particular marked flow. This comes at a cost of less reliable detection, since the single-message context creates more opportunities to detect errors. Our attacks are designed to work in both single-message and multiple-message contexts.

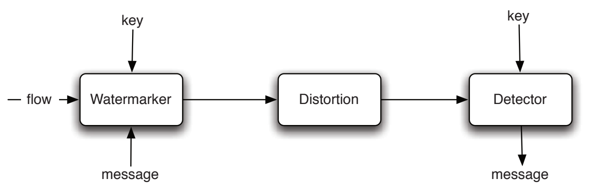


Figure 1: Network Flow Watermarking

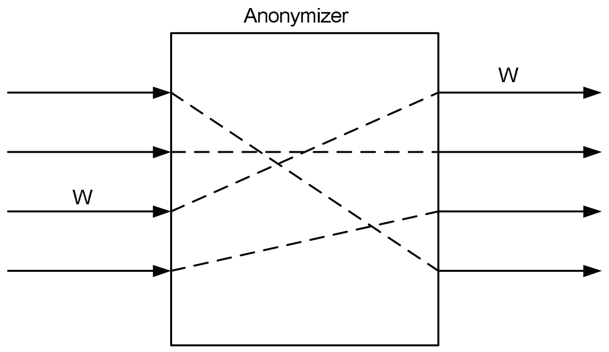


Figure 2: An anonymous system.

### 2.2 Watermarks in Anonymous Systems

At a very high level, an anonymous system maps a number of input flows to a number of output flows while hiding the relationship between them, as shown in Figure 2. The internal operation can be implemented by a mix network [8], onion routing [23], or a simple proxy [6]. The goal of an attacker, then, is to link an incoming flow to an outgoing flow (or vice versa).

A watermark can be used to defeat anonymity protection by marking certain input flows and watching for marks on the output flows. For example, a malicious website might insert a watermark on all flows from the site to the anonymizing system. A cooperating attacker who can eavesdrop on the link between a user and the anonymous system can then determine if the user is browsing the site or not. Similarly, a compromised entry router in Tor [11] can watermark all of its flows, and cooperating exit routers or websites can detect this watermark.

Note that this does not enable a fundamentally new attack on low-latency anonymous systems: it has been long known [23] that an attacker who can observe a flow at two points can determine if the flow is the same, unless cover traffic is used. (In fact, deployed low-latency systems such as Onion Routing [23], Freedom [1], and Tor [11] have all opted to forego cover traffic due to it being expensive, hoping instead that it will be difficult for an attacker to observe a significant fraction of incoming and outgoing flows.) However, watermarking makes the attack much more efficient. With passive traffic analysis, if one attacker observes $n$ input flows and another observes $m$ output flows, the attack will require $O(n)$ communication between the attackers and $O(nm)$ computation, as one attacker must transmit characteristics of all $n$ flows to the other, and then each output flow must be matched against each input flow. With watermarking, on the other hand, no communication needs to take place between the two attackers after they have established a shared secret key, and the computation cost is $O(n)$ and $O(m)$ at the watermarker and detector respectively, as the watermarker marks each input flow and the detector checks each output flow for the presence of a mark.

**Multi-Flow Attack** In the above examples, a website or an input router will insert the watermark into all the input flows going through them. Therefore, it will be possible for the anonymous system to obtain multiple watermarked flows. These flows can then be used to recover the secret key and then remove the watermarks from subsequent flows, using the techniques we describe below. Our techniques are low-cost, requiring a small number of watermarked flows and modest computation, so it is easy to check whether watermarking is being applied by a given website or router by aggregating its flows.

The only context where our attack does not apply is in a *traffic confirmation attack*. In this case, an attacker already has a strong suspicion that a particular input flow corresponds to a particular output flow, and therefore need only watermark a single flow. Traffic confirmation attacks are a more rare use of traffic analysis, since they only confirm existing suspicions, rather than revealing new linkages between flows. Furthermore, the efficiency gains of watermarks are not beneficial in this case, since $n = m = 1$. Therefore, our attack will apply to the vast majority of practical uses of watermarks in anonymous systems.

## 2.3 Watermarks in Stepping Stones

A stepping stone is a host that is used to relay traffic through an enterprise network to another remote destination, in order to hide the true origin of the flow. To detect such hosts, an enterprise must be able to link an incoming flow to the relayed outgoing flow. The situation is therefore very similar to an anonymous communication system, with $n$ flows entering the enterprise and $m$ flows leaving. Once again, this task may be accomplished by

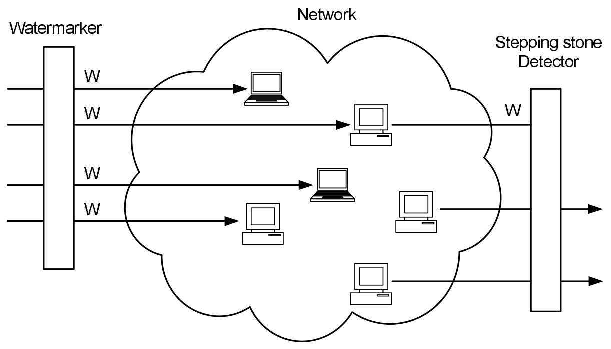


Figure 3: Stepping stone detection architecture.

passive traffic analysis [26, 29, 5, 12], but watermarks make such detection much more efficient. Passive techniques will require $O(nm)$ computation and potentially $O(n)$ communication, if there are multiple border routers through which traffic can enter or leave the enterprise. With watermarking, border routers for an enterprise will insert watermarks on all incoming flows, and check for the presence of the mark on all outgoing flows, as shown in Figure 3, reducing the computation cost to $O(n)$ and $O(m)$ for the incoming and outgoing flows.

**Multi-Flow Attack** Since all incoming flows must be marked, an attacker in control of a compromised host can simply generate multiple external flows destined for that host (and not relay them), and then collect the timing characteristics of the flows as they arrive at the host to recover the secret watermark key. Once this is accomplished, the key can be used to remove watermarks from relayed flows, thus defeating stepping stone detection.

## 2.4 Interval Centroid-Based Watermarking (ICBW)

We next review the scheme proposed by Wang et al. [25]; for more details of the scheme as well as some analysis we refer the reader to [25]. The scheme is based on dividing the stream into intervals of equal lengths, using two parameters: $o$, the offset of the first interval, and $T$, the length of each interval. A subset of $2n = 2rl$ of these intervals are chosen at random, and then randomly divided into two further subsets $A$ and $B$ each consisting of $n = rl$ intervals. Each of the sets $A$ and $B$ are randomly divided to $l$ subsets denoted by $\{A_i\}_{i=1}^{l}$ and $\{B_i\}_{i=1}^{l}$, each consisting of $r$ intervals. The $i$-th watermark bit is encoded using the sets $\{A_i, B_i\}$. Therefore, a watermark of length $l$ can be embedded in the flow. Figure 4 depicts the random selection and grouping of time intervals within a flow for watermark insertion.

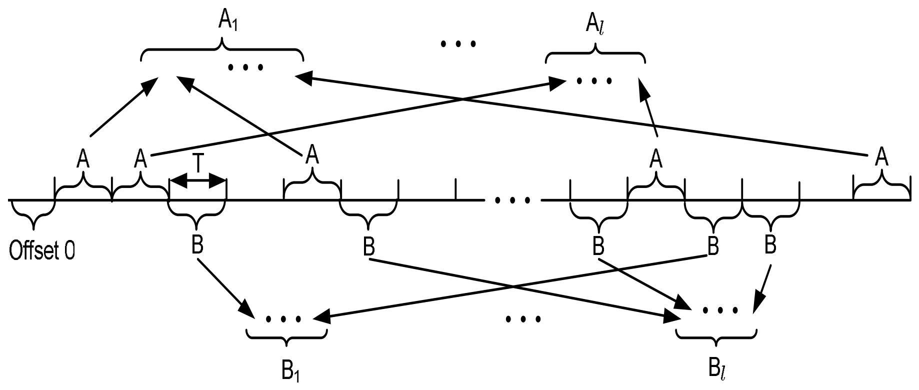


Figure 4: Random selection and assignment of time intervals within a packet flow for watermark insertion.

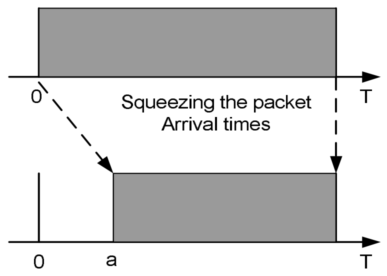


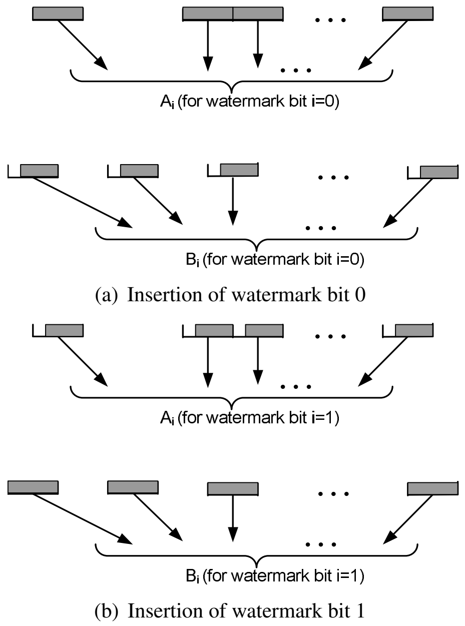
Figure 5: Distribution of packet arrival times in an interval of size $T$ before and after being delayed.

The watermarker and detector agree on the parameters $o$, $T$ and use a random number generator (RNG) and a seed $s$ to randomly select and assign intervals for watermark insertion. To keep the watermark transparent, all of these parameters are kept secret. Depending on whether the $i$-th watermark bit is 1 or 0, the watermarker delays the arrival times of the packets at the interval positions in sets $A_i$ or $B_i$ respectively, by a maximum of $a$. Figure 5 illustrates the effect of this delaying strategy over the distribution of packet arrival times in an interval of size $T$ (this operation is called "squeezing" by Wang et al.) Finally, the overall watermark embedding is illustrated in Figures 6 (a) and (b).

As the result of this embedding scheme, the expected value of aggregate centroid, i.e., the average offset of the packet arrival time from the beginning of the current length $T$ interval, in either the intervals $A_i$ (when watermark bit is 1) or $B_i$ (when watermark bit is 0) corresponding to bit $i$ is increased by $\frac{a}{2}$. The difference between the aggregate centroid of $A_i$ and $B_i$ now will be $\frac{a}{2}$ when watermark bit is 1 or $-\frac{a}{2}$ when watermark bit is 0.

The detector checks for the existence of the watermark bits. The check on watermark bit $i$ is performed by testing whether the average difference of the aggregate centroid of packet arrival times in the intervals $A_i$ and $B_i$ is closer to $\frac{a}{2}$ or $-\frac{a}{2}$. If it is closer to $\frac{a}{2}$, then the watermark bit is decoded as 1 and if it is closer to $-\frac{a}{2}$, the bit is declared a 0. By focusing on the arrival times of many intervals ($r$ of them for each bit of the watermark) rather than individual packet timings, the ICBW approach is robust to repacketization, insertion of chaff, and mixing of data flows. Network jitter can shift packets from one interval into another, but the suggested parameters for $a$ and $T$ (350ms and 500ms respectively) are large enough that few packets will be affected.

The secrecy of the interval positions $A_i$ and $B_i$ make the mark difficult to detect or remove, as it is hard to dis-



Ai (for watermark bit i=0)
Bi (for watermark bit i=0)
(a) Insertion of watermark bit 0
Ai (for watermark bit i=1)
Bi (for watermark bit i=1)
(b) Insertion of watermark bit 1

Figure 6: ICBW bit insertion

tinguish the patterns generated by the mark from natural variation in traffic rates. We show in Sections 3 and 4, however, that a simple technique allows an observer to effectively recover the watermark positions and values. This technique is applicable to any watermarking scheme that creates periods of clear or low traffic at *specific* parts of the flows across many flows. Next, we briefly describe *Interval-Based Watermarking (IBW)*, a flow watermarking scheme proposed by Pyun et al. [21] to detect *stepping stones*. Our attacks also applies to this scheme.

## 2.5 Interval-Based Watermarking

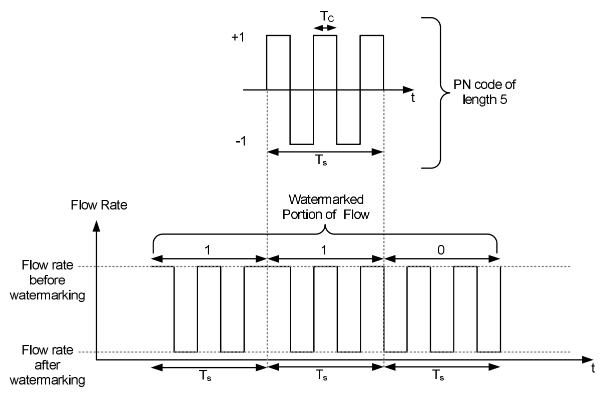
Similar to ICBW, the watermarking scheme of Pyun et al. [21] manipulates the arrival times of the packets over a set of preselected intervals. The watermark embedding is achieved by manipulating the rates of traffic in successive intervals. There are two manipulations: an interval $I_i$ may be *cleared* by delaying all packets from interval $I_i$ until interval $I_{i+1}$, or it may be *loaded* by delaying all packets from interval $I_{i-1}$ until interval $I_i$. A loaded interval will therefore have twice the expected number of packets, and a cleared one will have none. To send a 0 bit in position $i$, the interval $I_i$ is cleared and $I_{i+1}$ is loaded; to send a 1, $I_i$ is loaded and $I_{i+1}$ is cleared. (Note that since clearing one interval implicitly loads the next, it takes 3 intervals to send a bit.)

The watermarker and detector agree on the parameters $o, T$ and a list of positions $S = \{s_1, \ldots, s_n\}$; all of these parameters are secret. The watermarker encodes the watermark bits at the interval positions $s_i$ and the detector checks for the existence of the watermark. The check is performed by testing whether the data rate in interval $I_{s_i}$ differs from the rate in interval $I_{s_i+1}$ by a factor exceeding a threshold; if it does, then a 0 or 1 bit is considered detected. By focusing on data rates rather than individual packet timings, the interval-based approach is robust to repacketization of data flows.

The detection process may generate false positives due to natural variation in packet rates, or false negatives, as delays between the watermarker and repacketization at the relay cause rates in intervals to shift. To ensure reliable transmission, each watermark bit is encoded in several positions in the stream. Pyun et al. show that this technique operates with very low false positive and false negative rates.

## 2.6 Spread-Spectrum Watermarking

In the DSSS watermarking technique due Yu et al. [28], a binary watermark is embedded in the flow to achieve *invisible* traceback. In their proposed approach, each bit of a length $n$ binary watermark is embedded in an interval of length $T_s$. Hence the whole watermark is inserted



Tc
+1
-1
t
Ts
PN code of length 5
Flow Rate
Watermarked Portion of Flow
1
1
0
Flow rate before watermarking
Flow rate after watermarking
Ts
Ts
Ts
t

Figure 7: A length-5 PN code and insertion of DSSS watermark 110.

in an interval of length $nT_s$. To embed a watermark bit 1, the rate of the packets in the designated interval of length $T_s$ are manipulated according to a Pseudo-Noise (PN) code. The PN code is a quickly varying signal that switched between $+1$ and $-1$ and duration of each $\pm 1$ period is $T_c$. In particular, Yu et al. [28] choose a length-7 PN code for their implementation. When PN code is $+1$, the rate of the flow remains intact, but when PN code is $-1$, the rate of the flow is decreased for a duration of $T_c$.[2] On the other hand, to embed a watermark bit 0, the flow is manipulated using the complement of the PN code. Figure 7 depicts the embedding of watermark 110 for a PN code of length 5.

The watermarker and detector agree on the parameter $T_s$ and a Pseudo-Noise code. The detector recovers the watermark by first applying a high-pass filter to the received signal and subsequently passing it through despreading and a low-pass filter. The details of the detector's structure are inconsequential to our attack and the interested reader is referred to [28].

Given that the watermark insertion technique in DSSS reduces the flow rates over certain intervals across all flows, it is vulnerable to our multi-flow attack.

## 3 Attack Analysis

In this section, we present a probabilistic analysis of our attack using a model for interactive traffic. Though some watermarked traffic may consist of non-interactive bulk transfer traffic, we will show in Section 4.1 that interactive traffic presents a more difficult case for our attack, and thus we analyze it here. As DSSS watermarks work well only against non-interactive traffic, our analysis here applies only to IBW and ICBW, but as we demonstrate experimentally, our attack will work on DSSS watermarks as well.

## 3.1 Model of Interactive Traffic

We first present a model for interactive traffic, as it is essential to our analysis. Let $f_m$ denote the $m$-th flow in a pool of interactive traffic flows. Given that the traffic might be encrypted, we do not consider the content of the packets; likewise, the sizes of packets representing keystrokes are likely to be uniform. We thus consider only the arrival time of the packets in the flow, allowing us to model the flow as a point process.

Suppose we observed packet arrivals at times $t_1 < t_2 < \cdots < t_n$ in a fixed interval $(0, \tau]$ such that $t_i$ is the time the $i$-th packet arrived. The collection of arrival times $\mathbf{t}_m = (t_1, t_2, \ldots, t_n)$ specifies a flow $f_m$. Furthermore, we model the interactive connection as a Markov-modulated Poisson process (MMPP) [14, 15]. The set of possible states are $\{0, 1\}$, where state 0 corresponds to user typing characters and state 1 corresponds to periods of silence. Figure 8 depicts this two-state MMPP.

Let $X(t)$ denote the state of the process at time $t$. When the process is in state 0, packet arrivals are modeled as a renewal process; i.e. the interarrival times are independent and identically distributed (i.i.d.). In case of interactive traffic flow, this renewal process is often modeled as Poisson [12, 5]. The Poisson assumption means that the interarrival times of the packets, denoted by $\theta$, are exponentially distributed. Hence their probability density function (PDF) is given by:

$$f_\theta(t) = \lambda e^{-\lambda_0 t}$$

where $\lambda_0$ denotes the rate of the Poisson process. When the process is in state 1, the arrivals are again modeled as Poisson but with rate $\lambda_1 < \lambda_0$. Given that state 1 corresponds to a period of silence (no packet arrivals), as soon as a packet arrives, the embedded Markov chain transitions to state 0. Therefore, the transition probabilities $\{P_{ij}, i, j = 0, 1\}$ of the embedded Markov chain $\{X_n, n \geq 0\}$ are as follows:

$$\begin{aligned} P_{00} + P_{01} &= 1, \\ P_{01} = 1, P_{11} &= 0 \end{aligned} \tag{1}$$

and the embedded Markov chain is defined by the matrix:

$$\begin{bmatrix} P_{00} & 1 \\ 1 - P_{00} & 0 \end{bmatrix}$$

The steady state probabilities $\pi_0, \pi_1$ of the embedded chain $X_n$ are given by:

$$\begin{bmatrix} \pi_0 \\ \pi_1 \end{bmatrix} = \begin{bmatrix} P_{00} & 1 \\ 1 - P_{00} & 0 \end{bmatrix} \begin{bmatrix} \pi_0 \\ \pi_1 \end{bmatrix}$$

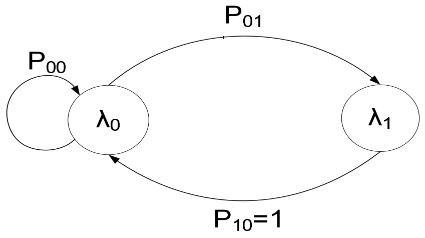


Figure 8: The embedded two-state Markov chain.

or:

$$\pi_0 = \frac{1}{2 - P_{00}}, \qquad \pi_1 = \frac{1 - P_{00}}{2 - P_{00}}$$

The steady state probabilities $P_0, P_1$ of the Markov process $X(t)$ are given by [15]:

$$P_i = \frac{\frac{\pi_i}{\lambda_i}}{\sum_k \frac{\pi_k}{\lambda_k}}$$

or:

$$P_0 = \frac{\lambda_1}{\lambda_1 + (1 - P_{00})\lambda_0}, P_1 = \frac{(1 - P_{00})\lambda_0}{\lambda_1 + (1 - P_{00})\lambda_0} \tag{2}$$

The significance of the steady state probabilities of (2) is that they capture the probability of each of the states 0 and 1 at any given point in time. Recall that ICBW encodes the watermark bits "1" or "0" by delaying the arrival times of the packets in the set of intervals $A_i$ or $B_i$ respectively and IBW encodes the watermark bits "0" or "1" by transferring the traffic of an interval of length $T$ to some adjacent interval. Therefore, they both create periods of times with no arrivals in the flow. This period for ICBW is of length $a$ and for IBW is of length $T$. When the embedded Markov chain is in state $i$, we can compute the probability of zero occurring in a period of length $\ell$ starting at any given point as:

$$P_{f_m^i}(0; \ell) = e^{-\lambda_i \ell} \tag{3}$$

since the waiting times are exponentially distributed and therefore memoryless.

In general, given a flow $f_m$ generated from an MMPP, from (3), the probability of having a period of length $\ell$ with no arrivals $P_{f_m}(0; \ell)$ is:

$$\begin{aligned} P_{f_m}(0; \ell) &= P_0 P_{f_m^0}(0; \ell) + P_1 P_{f_m^1}(0; \ell) \\ &= P_0 e^{-\lambda_0 \ell} + P_1 e^{-\lambda_1 \ell} \end{aligned} \tag{4}$$

where the steady state probabilities $\{P_0, P_1\}$ are given by (2).

A good watermarking scheme requires that the watermarked stream should not reveal any clues of the presence of the watermark to unauthorized observer. Therefore, it is desirable to pick $\ell$ such that $P_{f_m}(0; \ell)$ above should be reasonably large, so that presence of silent periods does not give away the watermark. We next present parameters of our two-state MMPP and show that, for those parameters, the watermark indeed cannot be detected by observing a single stream watermarked with ICBW or IBW. However, we will show that if attackers have access to multiple copies of a marked signal, they can defeat the two watermarking schemes both when multiple flows are watermarked with the same message and when different messages are embedded in different flows.

## 3.2 Parameter Selection and Goodness of Fit

We estimated the parameters $P_{00}$, $\lambda_0$, and $\lambda_1$ of our MMPP model by using network traces of SSH connections taken at a wireless access point in our institution. For a trace, we first estimated the underlying state of the embedded Markov chain by choice of a threshold $\eta$. If the interarrival time between two packets exceeded the threshold $\eta$, we assumed that the process was in state 1 and if the interarrival time between two packets was less than the threshold $\eta$, we assumed that the user was typing and therefore the process was in state 0. Once the states $\{X_n, n \geq 0\}$ of the underlying chain are determined, by concatenation of the parts of the interactive traffic that came from same underlying state, we could extract two Poisson flows with rates $\lambda_0$ and $\lambda_1$ from the original flow.

Given that the expected number of arrivals of a Poisson process distribution with parameter $\lambda$ in time interval $(0, t]$ is $\lambda t$, we estimated the rates $\lambda_0$ and $\lambda_1$ by calculating the arrival rates of each of the two extracted flows. Parameter $P_{00}$ was estimated as the portion of the time the chain spent at state 0. Our estimated values for the transition probability $P_{00}$ and the rates $\lambda_0$ and $\lambda_1$ were as follows:

$$P_{00} = .96 \qquad \lambda_0 = 5.6 \qquad \lambda_1 = 0.57 \tag{5}$$

To assess the goodness of fit of our MMPP model with parameters of (5), we used a quantile–quantile (q–q) plot [7]. Using the theoretical CDF of the model, the observations are mapped into values in interval $[0, 1]$. If the underlying statistical model of the data is consistent with the observations, the values obtained from the mapping are uniformly distributed in the interval $[0, 1]$. To assess the uniformity of the mapped values or equivalently assessing the goodness of the theoretical model an empirical CDF of the mapped values is compared against the theoretical CDF of a uniform distribution, which is a 45-degree reference line. The closer the CDF to this reference line, the greater the evidence that the statistical model captures the underlying phenomenon. The q–

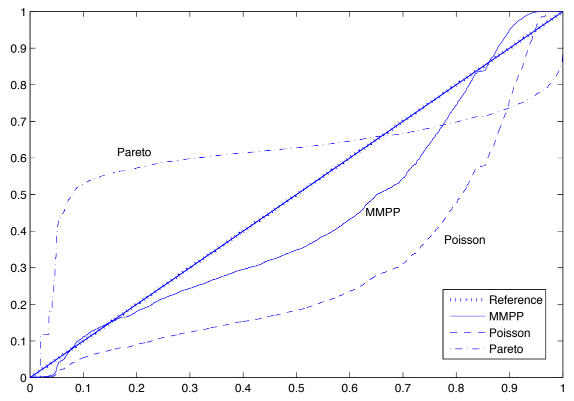


Figure 9: Q–Q plot of Poisson and MMPP models with our sample data.

q plot in Figure 9 shows that the MMPP model for the interactive traffic with parameters (5) provides a good fit for the data and significantly outperforms a simpler Poisson model, or a Pareto distribution that has been previously proposed to fit interactive traffic [19].

## 3.3 Multi-Flow Attack

Regardless of whether the ICBW or IBW watermarking schemes are implemented using the same message across all interactive flows or they use multiple messages for different flows, they are subject to an averaging attack. This is because both schemes embed watermarks by emptying the same parts across various flows. Next, we will explain our attack for both the single-message and multiple-message watermarks.

### 3.3.1 Single-Message Watermarks

When ICBW or IBW watermarking schemes are implemented using the same message across all interactive flows, an attacker who has access to $k$ watermarked flows can form an aggregate of all the flows, taking the sorted union of all the arrival times of packets in all flows. We denote this aggregated stream by $\overline{f_k}$, where the subscript $k$ denotes the number of streams involved in forming the aggregate flow.

Given that each interactive stream is independent of all the other streams, the probability of having a period of length $T$ with no arrivals in the flow $\overline{f_k}$ is given by:

$$\begin{aligned} P\{N_{\overline{f_k}}(t_a + \ell) - N_{\overline{f_k}}(t_a) = 0\} &= \prod_{i=1}^{k} P_{f_i}(0; \ell) \\ &= P_{f_m}(0; \ell)^k \quad (6) \end{aligned}$$

Equation (6) shows that probability of having period of length $\ell$ with no arrivals decreases exponentially in $k$, the number of copies used to form the aggregate flow $\overline{f_k}$. Therefore, if the streams are not watermarked there is a very small probability that the aggregate stream has periods of no arrivals. However, if both ICBW and IBW use the same key and message across all interactive flows, the aggregated copy of the watermarked flows always exhibits patterns of no arrivals of length $\ell$ that give away the location of the watermark as well as the maximum delay parameter $a$ of ICBW and the period $T$ of IBW.

Substituting the parameters of (5) into (4), assuming $\ell = 350ms$, as suggested by Wang et al. [25], we have $P_{f_m}(0; 0.35) = 0.33$. Therefore, in an aggregate of as few as 10 flows probability of a period of 350ms without any arrivals is as low as $P_{f_m}(0; 0.35)^{10} = 1.6 \times 10^{-5}$. Similarly, for $\ell = 900ms$, as used by Pyun et al. [21], we have $P_{f_m}(0; 0.9) = 0.17$ and $P_{f_m}(0; 0.9)^{10} = 2.4 \times 10^{-8}$.

### 3.3.2 Multi-Message Watermarks

If different flows are used to encode different messages, simple aggregation will no longer work, since by switching between 1 and 0 bits, both ICBW and IBW apply different transforms to different intervals. For example, with ICBW, a given interval may be squeezed when a certain bit is 0, and not squeezed when that bit is 1. By aggregating flows where that bit changes, no empty periods will be detected.

However, by observing a few more flows, we can still detect the presence of a watermark. Given a bit $b$ and a set of $2k-1$ flows, by the pigeon hole principle, there exists a subset of $k$ flows where the bit has the same value. If we aggregate all the flows in that subset, we will find clear intervals of length $a$ or $T$, depending on the scheme that is used.

To detect the watermark, then, we examine all $\binom{2k-1}{k}$ subsets of $k$ flows out of a collection of $2k - 1$. For each bit position, we will be able to find at least one subset where that bit value is all the same, and we can thus detect it with the same ease as when a single-valued watermark is used. The number of subsets is, of course, superexponential in $k$, but our attack works with values of $k$ around 10, making such a search feasible, as $\binom{19}{10} = 92378$.

Examining all these subsets increases the possibility of a false positive—a naturally occurring cleared interval in the aggregate flow. However, such false positives will be relatively rare, so the attacker can estimate the value of $\ell$ and then discard intervals that do not match.

## 3.4 Impact of Timing Perturbations

Our analysis so far has assumed that the attacker sees the timings of the watermarked stream directly. In reality, these timings will be perturbed by network delays. As a result, the intervals cleared by the watermark may have some packets from previous intervals shifted into them and no longer appear completely empty. Note that what is relevant here is not the magnitude of the network delay but its variance, or *jitter*, since delaying all packets by an equal amount does not affect our attack. And if the jitter is much less than $\ell$, our attack will work equally well: if jitter is $< \epsilon$ with high probability, then we will find clear intervals of length at least $\ell - \epsilon$ in the $k$ aggregated watermarked streams, whereas the probability of seeing such an interval in unwatermarked streams is $P_{f_m}(0; \ell-\epsilon)^k \approx P_{f_m}(0; \ell)^k$, which is vanishingly small. We observe that the studied parameters of the ICBW and IBW schemes have $\ell$ = 350ms or 900ms, in order to resist traffic perturbations, repacketization, etc. The network jitter, on the other hand, is two orders of magnitude smaller. Our experiments on PlanetLab [3] show it to be on the order of several milliseconds for geographically distributed hosts, and this matches the results of previous studies [18]. Therefore, it is indeed the case that the jitter is $< \epsilon \ll \ell$, so it will not significantly affect our attack.

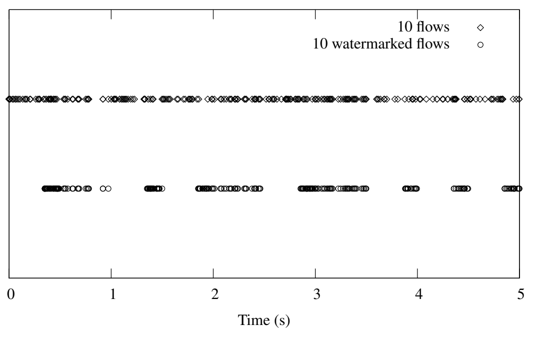
## 4 Implementation

Having shown the theoretical background behind our attack, we now show the result of implementing it in practice. We developed algorithms to detect the presence of a watermark, recover the secret parameters, and to remove the watermark from new streams. We evaluated the algorithms using both real flows gathered from traces and synthetic flows generated using our MMPP model, presented in Section 3.1. We first present our attacks for single-message watermarks, and then extend it to watermarks that use multiple messages.

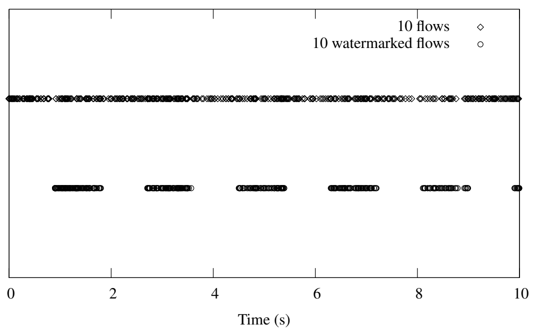
## 4.1 Watermark Detection

As above, our attack relies on collecting a series of flows that are watermarked with the same message. These flows are combined into a single flow and examined for large gaps between packets. Figure 10(a) shows the packet arrivals for 10 combined flows before and after an ICBW watermark has been applied. The watermark pattern is clearly visible in the combined flows, revealing the presence of a watermark. Figure 10(b) shows the same process working with the IBW watermark scheme.

We also performed the same analysis for non-interactive, bulk transfer traffic by applying the watermark to packet traces we collected from web downloads

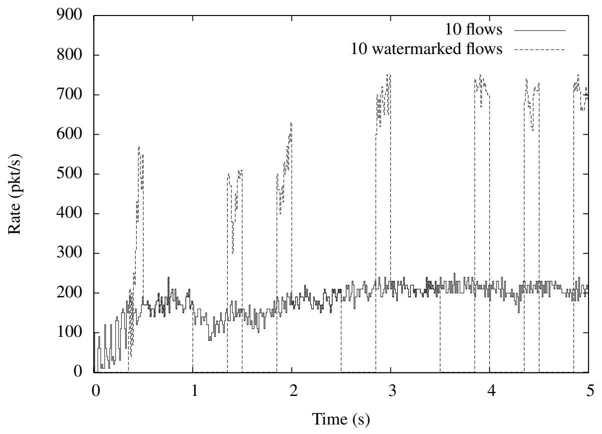


(a) Interval-Centroid Based Watermark

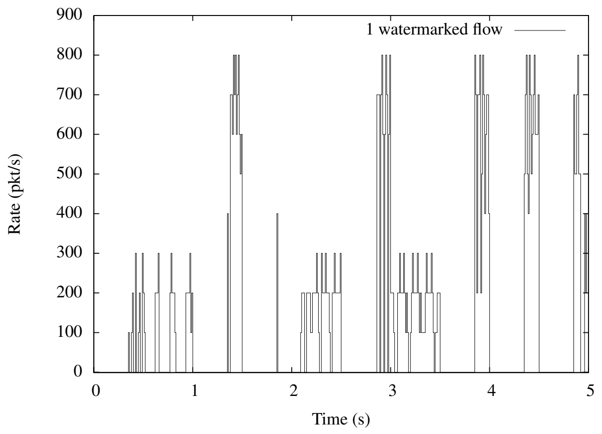


(b) Interval-Based Watermark

Figure 10: 10 flows before and after watermarking.



(a) Watermark on 10 flows



(b) Watermark on 1 flow

Figure 11: Watermark detection on bulk traffic.

across a DSL connection. Figure 11(a) shows the packet timings for 10 combined flows before and after a watermark. Bulk transfers have a somewhat more regular behavior, since they are controlled by the TCP algorithms, rather than by individual users. This can be seen at the beginning of the 10 combined flows before watermark: the TCP slow start period results in a much lower rate for the first few seconds of the connection. However, this regularity quickly gets out of sync due to irregular network delay and response times. In the graph of 10 watermarked flows, the intervals squeezed by the watermark are readily visible. In fact, because data transfer flows are much more dense than interactive flows, the watermark is visible even on a single flow (Figure 11(b)).

The DSSS watermark is intended to be applied to bulk transfer traffic such as FTP, since it interferes with traffic rate, rather than changing packet timings. A similar multi-flow attack works against DSSS as well, as shown in Figure 12. (We used the parameters of chip length 0.4s, chip sequence length of 7, and code length of 7.) In this case, periods of high interference are clearly seen as low-rate periods in the flows, allowing one to recover the chip sequence and then decode the watermark.

## 4.2 Watermark Removal

Based on the combined graphs, it is easy to recover the watermark parameters as well. We can build a template of clear intervals by selecting all intervals larger than a threshold; for example, Figure 13(a) shows the template derived from 10 flows watermarked by ICBW. The estimated template is somewhat imprecise, due to network jitter, as well as the fact that small (10–20ms) gaps may precede or follow the clear intervals even when 10 flows are combined. However, this imprecision is not a problem since the watermark can still be effectively removed. The template also lets us estimate the values of $T$ and $a$. We can average the lengths of clear intervals and the distance between two consecutive clear intervals to obtain a relatively precise estimate. Armed with this information, we can then modify a new flow to remove the watermark.

For ICBW, we have two choices: we can either shift traffic into the clear intervals in the template, thereby negating the squeezing action of the watermark, or find

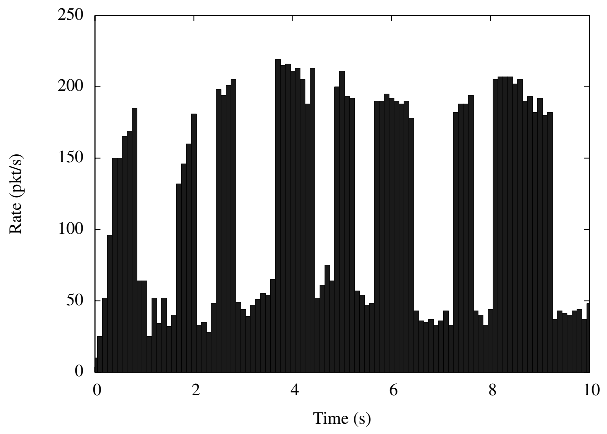
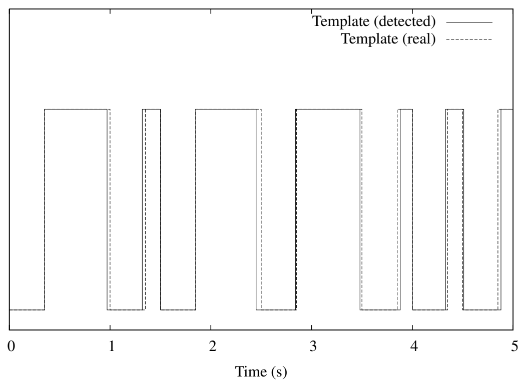


Figure 12: Average rate of 10 flows after DSSS watermark.

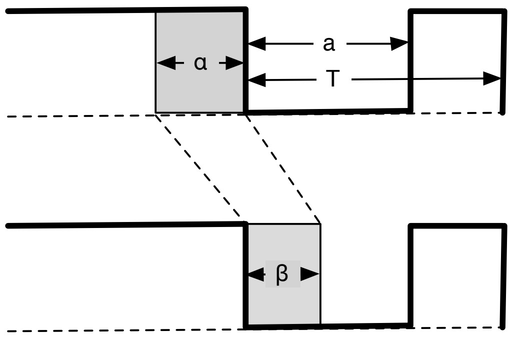
intervals that have not been squeezed and squeeze them. We decided to implement the former approach since it does not require as precise an estimate of $T$. Also, it leaves the flow looking more natural. Our shift is implemented as shown in Figure 13(b), by shifting all packets in a period $\alpha$ before the clear interval into an interval of length $\beta$ inside the clear interval. Larger values of $\alpha$ and smaller values of $\beta$ will more significantly shift the interval centroid back in a different direction; however, very small values of $\beta$ may not have the desired effect, since the template is imprecise and too many packets may get shifted without arriving into the correct interval. Experimentally, we found that $\alpha = 0.9(\hat{T} - \hat{a})$ and $\beta = 0.8(\hat{T} - \hat{a})$ provide best results, where $\hat{T}$ and $\hat{a}$ are estimated values of $T$ and $a$.

Table 1 shows the results of watermark removal. We reimplemented the ICBW detection mechanism and computed the Hamming distance of the encoded watermark to the detected one, collected over 100 flows. (We show the average distance, with range shown in parentheses). With as few as 10 flows, we are able to get a reasonably good estimate of $T$ and $a$ and remove the watermark in most cases—the ICBW detection scheme uses a Hamming distance threshold of 5–8 to decide when a watermark has been detected. With 15 flows, we get a more accurate template and estimate, and all 100 flows will clear the template.

A similar approach can be used to attack the IBW watermark; by delaying packets so that they fall into the clear intervals, the clear intervals become indistinguishable from loaded ones. Table 2 shows the effect of applying our attack on the IBW watermark, where 24 bits are encoded at different levels of redundancy. Even with a redundancy of 80, most bits are not recovered correctly. These results were obtained by using the code provided by the authors of [21].



(a) Template of clear intervals



(b) Shift to remove watermark

Figure 13: Watermark Removal

We expect a similar technique should work against DSSS watermarks; a template of low rates can be inferred from several flows. An attacker can then decrease rates in the non-interference section of the template by dropping packets, or increase the rate in the high-interference section by delaying packets into the template. We do not have experimental results for DSSS since the detection algorithm is fairly complex and we did not have access to an implementation of it.

## 4.3 Multiple Messages

So far we have assumed that the watermarks on all of the aggregated flows are the same. Here, we consider the case where each watermark uses different messages. As described in Section 3.3.2, we can still execute our attack by relying on the fact that within a collection of $2k - 1$ flows, for any given bit $b$, we can find $k$ flows where this bit has the same value.

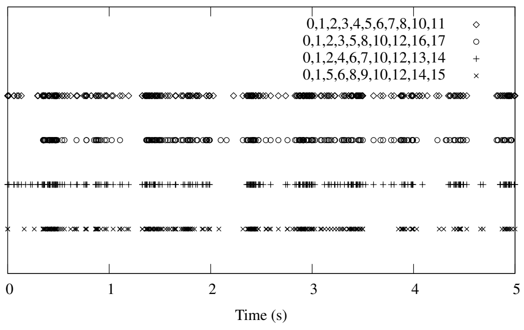
Figure 14(a) plots the result of such a subset search. By inspection, we can see that in the first subset of flows, the interval (4.5,4.85) has been cleared. In the second subset, this interval remains cleared and the interval (0,0.35) becomes clear as well. The third subset

Table 1: Results for removing ICBW watermarks

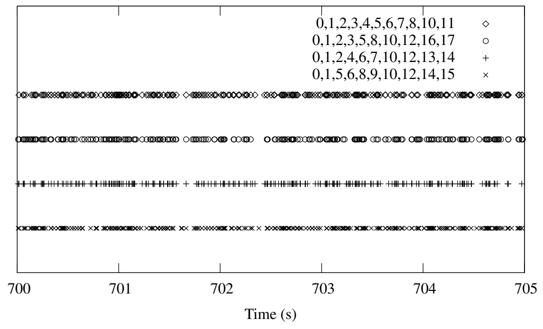
| **Num flows** | $\hat{a}$ | $\hat{T}$ | **Hamming not watermarked** | **Hamming watermarked** | **Hamming attacked** | **Ave. delay** | **Max delay** |
|---|---|---|---|---|---|---|---|
| 10 | 365 ($\sigma = 10.7$) | 492 ($\sigma = 15.2$) | 17.9 (13–24) | 2.67 (1–7) | 13.9 (2–20) | 33.6 | 164 |
| 15 | 353 ($\sigma = 0.60$) | 504 ($\sigma = 1.62$) | 17.6 (13–25) | 2.74 (0–6) | 16.1 (12–21) | 42.6 | 188.2 |
| 20 | 346 ($\sigma = 0.30$) | 504 ($\sigma = 0.50$) | 17.2 (12–21) | 2.68 (0–5) | 16.4 (11–20) | 45.4 | 194.3 |

Table 2: Watermark bits detected before and after applying the attack (watermark length is 24).

| **Rep.** | **Bits detected** | | **Marked packets** |
|---|---|---|---|
| | **Before attack** | **After attack** | |
| 1 | 7 | 3 | 53 |
| 5 | 14 | 5 | 156 |
| 10 | 24 | 4 | 505 |
| 15 | 24 | 2 | 754 |
| 20 | 24 | 2 | 967 |
| 24 | 24 | 2 | 1209 |
| 30 | 24 | 2 | 1440 |
| 35 | 24 | 2 | 1724 |
| 41 | 24 | 2 | 2008 |
| 45 | 24 | 2 | 2307 |
| 50 | 24 | 2 | 2697 |
| 55 | 24 | 2 | 3083 |
| 60 | 24 | 2 | 3296 |
| 65 | 24 | 2 | 3623 |
| 70 | 24 | 2 | 3876 |
| 75 | 24 | 2 | 4090 |
| 80 | 24 | 2 | 4343 |



(a) Watermarked flow subsets



(b) Un-watermarked flow subsets

Figure 14: Subset approach to multiple message watermarks

has no packets in (2.0,2.35) and the fourth in (3.5,3.85). Note that this pattern immediately lets us detect the presence of a watermark; Figure 14(b) shows the same flow subsets on an unwatermarked section.

Recovery of the secret parameters can proceed largely as in the single-message case. One difficulty is that with the flow subsets, we may encounter large intervals that are not precisely aligned with the interval positions. For example, Table 3(a) lists the blank intervals longer than 0.2s in the last subset. There are a lot of wrong-size intervals that result from the case when 8 or 9 of the flows in the subset have had an interval squeezed, but the last one or two add a few packets to the mix. To address this concern, we can select the largest empty intervals in any subset, as shown in Table 3(b). These will correspond to intervals that have been squeezed on every flow. This can be used to recover the watermark parameters of $T$ and $a$.

Once these are obtained, the next step is to scan through all subsets and determine which intervals are always squeezed at the same time and call such lists $S_i$; these will correspond to either $A_b$ or $B_b$ for some bit $b$. Then, for each $S_i$, we find $S_j$ such that $S_i$ and $S_j$ are never squeezed at the same time. This will tell us that $S_i$ and $S_j$ correspond to the same bit. Armed with this knowledge, we can remove the watermark by observing the watermarked stream for a short while, and when we see intervals from $S_i$ that are being squeezed, we proceed to artificially squeeze intervals in $S_j$ (or unsqueeze further intervals in $S_i$, or both).

Note that the subset technique can also be applied when not all the flows are watermarked. For example, a website may watermark only some connections that are

Table 3: Blank intervals from subset of flows

(a) All blank intervals

| Start | End |
|---|---|
| 2.08 | 2.32 |
| 3.50 | 3.85 |
| 4.03 | 4.25 |
| 5.13 | 5.33 |
| 11.59 | 11.85 |
| 18.14 | 18.37 |
| 19.56 | 19.79 |
| 25.58 | 25.82 |
| 30.06 | 30.34 |
| 34.08 | 34.35 |
| ... | ... |

(b) Largest blank intervals

| Start | End |
|---|---|
| 130.98 | 131.35 |
| 140.49 | 140.86 |
| 151.99 | 152.36 |
| 161.99 | 162.35 |
| 235.99 | 236.37 |
| 306.49 | 306.86 |
| 334.49 | 334.86 |
| 368.49 | 368.86 |
| 43.99 | 44.36 |
| 51.98 | 52.35 |
| ... | ... |

of particular interest; by finding subsets that are all watermarked, the mark can still be recovered. A scheme that probabilistically marked some flows and used different messages at the same time would present a challenge to our attack; however, we suggest that a different countermeasure be used, since it allows all flows to be marked, which is desirable for most applications.

## 5 Countermeasures

We next consider several countermeasures to our attack.

### 5.1 Multiple Offsets

The watermarking schemes we analyze have the ability to self-synchronize by trying different values for the offset $o$ and using the best match. Thus, $o$ can be changed for different streams. The synchronization mechanism can introduce more errors into the detection, but the use of increased encoding redundancy can make up for it.

The use of different offsets makes our attack more difficult, since simply aggregating $k$ flows will result in misalignment, destroying the clear intervals. It is, of course, possible to test different positions for $o$ for each stream, but to test $n$ positions in $k$ flows requires $n^{k-1}$ trials (we can hold the first flow fixed).

On the other hand, some alignments of two or three flows can be discarded immediately, if such an alignment results in few intervals that are clear of packets. Furthermore, the search for $o$ can be imprecise at first: even if each flow is aligned to within $0.1$s of the correct position, intervals of 150ms or 700ms will be seen in the average. Thus, changing offsets makes our attack more difficult, but not impossible to perform.

### 5.2 Multiple Positions

Another alternative is to choose different positions, in the case of ICBW and IBW, and different PN codes in the case of DSSS. Let us consider the case of ICBW. A watermarker and detector must use the same assignment of intervals to the sets $A_i$ and $B_i$, as determined by the random seed $s$, in order for the watermark to be successfully recovered. However, a watermarker may decide to use multiple seed values, $s_1, \ldots, s_n$, and pick one of them at random for each flow.

To deal with this, the detector would need to try to recover the watermark with each possible $s_i$ and pick the best match. Once again, the probability of error grows with $n$, but increased redundancy can again be used to make up for it. Note that the probability of error falls exponentially with increased redundancy, but grows only roughly linearly with $n$.

We can once again use the subset attack to try to find $k$ flows that use the same seed value $s_i$; however, the complexity grows quickly out of control. The probability of a given set of $k$ flows using the same seed is $\left(\frac{1}{n}\right)^{k-1}$, which falls quite quickly even when $k = 10$. By the pigeon hole principle, within $n(k-1)+1$ flows we can always find a subset of $k$ flows with the same seed, but the search space of all $\binom{n(k-1)+1}{k}$ subsets grows superexponentially in $n$. For example, with $n = 6$ and $k = 10$, $\binom{51}{10} > 10^{10}$, resulting in an infeasible number of subsets to enumerate.

The same principle can apply to IBW, by picking multiple sets of positions $\{s_i\}$, and to DSSS by using multiple PN codes.

## 6 Conclusion

We have demonstrated an attack on the interval centroid-based watermarking scheme and interval based watermarking scheme that is highly successful, while requiring a low amount of resources. Our attack is based on a solid theoretical grounding, and has been validated with a prototype implementation tested against the original ICBW and IBW prototypes. We can remove the watermark from an existing flow for both schemes. Additionally, in case of IBW we can also recover the watermark parameters and values, allowing us to modify the watermark or insert it into other streams, thereby confusing the detector. We have also suggested a countermeasure to our attack—switching bit positions. This countermeasure can impose a very high computation cost and therefore disable the attack.

While the use of network flow watermarking techniques for various security applications is quite new [27, 21, 25, 28], digital watermarking. and specifically multimedia watermarking is a nearly mature field. Indeed, most of network flow watermarking schemes are inspired by multimedia watermarks. To name a few,, Wang and Reeves's [27] scheme is a special instance of QIM watermarking, a well-understood multimedia watermarking technique [16]. The IBW scheme of Pyun et al. [21] is based on the patchwork watermark of Bender et al. [4] and the scheme of Yu et al. [28] is based on spread spectrum watermarking [9].

The current approach for designing network flow watermarks suffers from the fact that. while watermarking schemes are inspired by the digital watermarking schemes, little attention is given to the entirety of the watermarking design problem. For example, statistical characteristics of the underlying media are always an important consideration in digital watermarks, but network watermark research does not adequately model the effect that network traffic characteristics have on watermarks; as we showed, the density of bulk traffic makes it very difficult to insert a transparent watermark. Likewise, digital watermarks have long considered the possibility that multiple watermarked documents can be used to attack watermarks [10, 17], but we are unaware of previous work looking at the multi-flow threat model for watermarking. We thus hope that future work on watermarks will be informed by our work and perform a broader analysis.

## Acknowledgments

We would like to thank Peng Ning and Douglas Reeves for providing us with the IBW implementation, and Xinyuan Wang and the anonymous referees for providing feedback on earlier versions of this paper. This research was supported in part by NSF grants CNS–0627671 and CCF–0729061.

## References

[1] Back, A., Goldberg, I., and Shostack, A. Freedom systems 2.1 security issues and analysis. White paper, Zero Knowledge Systems, Inc., May 2001.

[2] Back, A., Möller, U., and Stiglic, A. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Information Hiding Workshop* (Apr. 2001), I. S. Moskowitz, Ed., vol. 2137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 245–247.

[3] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. Operating systems support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation* (Mar. 2004), R. Morris and S. Savage, Eds., USENIX, pp. 253–266.

[4] Bender, W., Gruhl, D., Morimoto, N., and A.Lu. Techniques for data hiding. *IBM Systems Journal 35*, 3/4 (1996), 313–336.

[5] Blum, A., Song, D. X., and Venkataraman, S. Detection of interactive stepping stones: Algorithms and confidence bounds. In *International Symposium on Recent Advances in Intrusion Detection* (Sept. 2004), E. Jonsson, A. Valdes, and M. Almgren, Eds., vol. 3224 of *Lecture Notes in Computer Science*, Springer, pp. 258–277.

[6] Boyan, J. The anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine 4*, 9 (September 1997).

[7] Brown, E. N., Barbieri, R., Ventura, V., Kass, R. E., and Frank, L. M. The time-rescaling theorem and its application to neural spike train data analysis. *Neural Computation 14*, 2 (2002), 325–346.

[8] Chaum, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (February 1981), 84–90.

[9] Cox, I., Kilian, J., Leighton, T., and Shamoon, T. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing 6*, 12 (1997), 1673–1687.

[10] Cox, I. J., Killian, J., Leighton, T., and Shamoon, T. Secure spread spectrum watermarking for images, audio, and video. In *IEEE International Conference on Image Processing* (1996), vol. III, pp. 243–246.

[11] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *USENIX Security Symposium* (Aug. 2004), M. Blaze, Ed., USENIX Association, pp. 303–320.

[12] Donoho, D., Flesia, A., Shankar, U., Paxson, V., Coit, J., and Staniford, S. Multiscale stepping-stone detection: detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *International Symposium on Recent Advances in Intrusion Detection* (Oct. 2002), A. Wespi, G. Vigna, and L. Deri, Eds., vol. 2516 of *Lecture Notes in Computer Science*, Springer, pp. 16–18.

[13] Federrath, H., Ed. *International Workshop on Design Issues in Anonymity and Unobservability* (July 2000), vol. 2009 of *Lecture Notes in Computer Science*, Springer.

[14] Fischer, W., and Meier-Hellstern, K. The Markov-modulated Poisson process (MMPP) cookbook. *Performance Evaluation 18*, 2 (1993), 149–171.

[15] Gallager, R. G. *Discrete Stochastic Processes*. Kluwer Academic Publishers, 1996.

[16] Goteti, A. K., and Moulin, P. QIM watermarking games. In *IEEE International Conference on Image Processing* (2004), pp. 717–720.

[17] Kilian, J., Leighton, F., Matheson, L., Shamoon, T., Tarjan, R., and Zane, F. Resistance of digital watermarks to collusive attacks. In *IEEE International Symposium on Information Theory* (1998), p. 271.

[18] Marsh, I., and Li, F. Wide area measurements of VoIP quality. In *Workshop on Quality of Future Internet Services* (2003).

[19] Paxson, V., and Floyd, S. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking 3*, 3 (June 1995), 226–244.

[20] Pfitzmann, B., and McDaniel, P., Eds. *IEEE Symposium on Security and Privacy* (May 2007).

[21] Pyun, Y., Park, Y., Wang, X., Reeves, D. S., and Ning, P. Tracing traffic through intermediate hosts that repacketize flows. In *IEEE Conference on Computer Communications (INFOCOM)* (May 2007), G. Kesidis, E. Modiano, and R. Srikant, Eds., pp. 634–642.

[22] Raymond, J.-F. Traffic analysis: Protocols, attacks, design issues, and open problems. In Federrath [13], pp. 10–29.

[23] Syverson, P., Tsudik, G., Reed, M., and Landwehr, C. Towards an analysis of onion routing security. In Federrath [13], pp. 96–114.

[24] Wang, X., Chen, S., and Jajodia, S. Tracking anonymous peer-to-peer VoIP calls on the Internet. In *ACM Conference on Computer and Communications Security* (Nov. 2005), C. Meadows, Ed., ACM, pp. 81–91.

[25] Wang, X., Chen, S., and Jajodia, S. Network flow watermarking attack on low-latency anonymous communication systems. In Pfitzmann and McDaniel [20], pp. 116–130.

[26] Wang, X., Reeves, D., and Wu, S. F. Inter-packet delay based correlation for tracing encrypted connections through stepping stones. In *European Symposium on Research in Computer Security* (Oct. 2002), D. Gollmann, G. Karjoth, and M. Waidner, Eds., vol. 2502 of *Lecture Notes in Computer Science*, Springer, pp. 244–263.

[27] Wang, X., and Reeves, D. S. Robust correlation of encrypted attack traffic through stepping stones by manipulation of inter-packet delays. In *ACM Conference on Computer and Communications Security* (2003), V. Atluri, Ed., ACM, pp. 20–29.

[28] Yu, W., Fu, X., Graham, S., D.Xuan, and Zhao, W. DSSS-based flow marking technique for invisible traceback. In Pfitzmann and McDaniel [20], pp. 18–32.

[29] Zhang, Y., and Paxson, V. Detecting stepping stones. In *USENIX Security Symposium* (Aug. 2000), S. Bellovin and G. Rose, Eds., USENIX Association, pp. 171–184.

## Notes

[1]We are unaware of a quantitative comparison of the accuracy of watermarking techniques with passive traffic analysis, but reported false-positive rates for most watermarking techniques are quite low. In any case, the two techniques can be combined to improve accuracy.

[2]Yu et al. suggest that this can be done by sending an interfering flow across a bottleneck link; their scheme is thus unique in not requiring full control of packet forwarding for the flow.

# Verifying Compliance of Trusted Programs

Sandra Rueda, Dave King and Trent Jaeger
*Systems and Internet Infrastructure Security Laboratory*
*Department of Computer Science and Engineering*
*The Pennsylvania State University*
{ruedarod,dhking,tjaeger}@cse.psu.edu

## Abstract

In this paper, we present an approach for verifying that trusted programs correctly enforce system security goals when deployed. A trusted program is trusted to only perform *safe* operations despite have the authority to perform *unsafe* operations; for example, initialization programs, administrative programs, `root` network daemons, etc. Currently, these programs are trusted without concrete justification. The emergence of tools for building programs that guarantee policy enforcement, such as security-typed languages (STLs), and mandatory access control systems, such as user-level policy servers, finally offers a basis for justifying trust in such programs: we can determine whether these programs can be deployed in compliance with the reference monitor concept. Since program and system policies are defined independently, often using different access control models, compliance for all program deployments may be difficult to achieve in practice, however. We observe that the integrity of trusted programs must dominate the integrity of system data, and use this insight, which we call the PIDSI approach, to infer the relationship between program and system policies, enabling automated compliance verification. We find that the PIDSI approach is consistent with the SELinux reference policy for its trusted programs. As a result, trusted program policies can be designed independently of their target systems, yet still be deployed in a manner that ensures enforcement of system security goals.

## 1 Introduction

Every system contains a variety of trusted programs. A *trusted program* is a program that is expected to *safely* enforce the system's security goals despite being authorized to perform *unsafe* operations (i.e., operations that can potentially violate those security goals). For example, the X Window server [37] is a trusted program because enables multiple user processes to share access to the system display, and the system trusts it to prevent one user's data from being leaked to another user. A system has many such trusted processes, including those for initialization (e.g., `init` scripts), administration (e.g., software installation and maintenance), system services (e.g., windowing systems), authentication services (e.g., remote login), etc. The SELinux system [27] includes over 30 programs specifically-designated as trusted to enforce multilevel security (MLS) policies [14].

An important question is whether trusted programs actually enforce the system's security goals. Trusted programs can be complex software, and they traditionally lack any declarative access control policy governing their behavior. Of the trusted programs in SELinux, only the X server currently has an access control policy. Even in this case, the system makes no effort to verify that the X server policy corresponds to the system's policy in any way. Historically, only formal assurance has been used to verify that a trusted program enforces system security goals, but current assurance methodologies are time-consuming and manual. As a result, trusted programs are given their additional privileges without any concrete justification.

Recently, the emergence of techniques for building programs with declarative access control policies motivates us to develop an automated mechanism to verify that such programs correctly enforce security goals. Programs written in security-typed languages [23, 26, 28] or integrated with user-level policy servers [34] each include program-specific access control policies. In the former case, the successful compilation of the program proves its enforcement of an associated policy. In the latter case, the instrumentation of the program with a policy enforcement aims to ensure comprehensive enforcement of mandatory access control policies. In general, we would want such programs to enforce system security goals, in which case we say that the program *complies* with the system's security goals.

We use the classical *reference monitor concept* [1] as the basis for the program's compliance requirements[1]: (1) the program policy must enforce a policy that represents the system security goals and (2) the system policy must ensure that the program cannot be tampered. We will show that both of these problems can be cast as policy verification problems, but since program policies and system policies are written in different environments, often considering different security goals, they are not directly comparable. For example, program policy languages can differ from the system policy language. For example, the security-typed language Jif [26] uses an information flow policy based on the Decentralized Label Model [24], but the SELinux system policy uses an extended Type Enforcement policy [5] that includes multi-level security labeling [2]. Even where program policies are written for SELinux-compatible policy servers [34], the set of program labels is often distinct from the set of system labels. In prior work, verifiably-compliant programs were developed by manually joining a system policy with the program's policy and providing a mapping between the two [13]. To enable general programs to be compliant, our goal is to develop an approach by which compliant policy designs can be generated and verified automatically.

As a basis for an automated approach, we observe that trusted programs and the system data upon which it operates have distinct security requirements. For a trusted program, we must ensure that the program's components, such as its executable files, libraries, configuration, etc., are protected from tampering by untrusted programs. For the system data, the system security policy should ensure that all operations on that data satisfy the system's security goals. Since trusted programs should enforce the system's security goals, their integrity must dominate the system data's integrity. If the integrity of a trusted program is compromised, then all system data is at risk. Using the insight that *program integrity dominates system integrity*, we propose the PIDSI approach to designing program policies, where we assign trusted program objects to a higher integrity label than system data objects, resulting in a simplified program policy that enables automated compliance verification. Our experimental results justify that this assumption is consistent with the SELinux reference policy for its trusted programs. As a result, we are optimistic that trusted program policies can be designed independently of their target systems, yet still be deployed in a manner that ensures enforcement of system security goals.

After providing background and motivation for the policy compliance problem in Section 2, we detail the following novel contributions:

1. In Section 3, we define a formal model for *policy compliance problem*.
2. In Section 4, we propose an approach called *Program Integrity Dominates System Integrity* (PIDSI) where trusted programs are assigned to higher integrity labels than system data. We show that compliance program policies can be composed by relating the program policy labels to the system policy on the target system using the PIDSI approach.
3. In Section 5.1, we describe policy compliance tools that automate the proposed PIDSI approach such that a trusted program can be deployed on an existing SELinux system and we can verify enforcement of system security goals.
4. In Section 5.2, we show the trusted programs for which there are Linux packages in SELinux are compatible with the PIDSI approach with a few exceptions. We show how these can be resolved using a few types of simple policy modifications.

This work is the first that we are aware of that enables program and system security goals to be reconciled in a scalable (automated and system-independent) manner.

## 2 Background

The general problem is to develop an approach for building and deploying trusted programs, including their access control policies, in a manner that enables automated policy compliance verification. In the section, we specify the current mechanisms for these three steps: (1) trusted program policy construction; (2) trusted program deployment; and (3) trusted program enforcement. We will use the SELinux system as the platform for deploying trusted programs.

### 2.1 Program Policy Construction

There are two major approaches for constructing programs that enforce a declarative access control policy: (1) security-typed languages [26, 28, 33] (STLs) and (2) application reference monitors [22, 34]. These two approaches are quite different, but we aim to verify policy compliance for programs implemented either way.

Programs written in an STL will compile only if their information flows, determined by type inferencing, are authorized by the program's access control policy. As a result, the STL compilation guarantees, modulo bugs in the program interpreter, that the program enforces the specified policy. As an example, we consider the Jif STL. A Jif program consists of the program code plus a program policy file [12] describing a Decentralized Label Model [24] policy for the program. The Jif compiler ensures that the policy is enforced by the generated program. We would use the policy file to determine whether Jif program complies with the system security goals.

For programs constructed with application reference monitors, the program includes a reference monitor interface [1] which determines the authorization queries that must be satisfied to access program operations. The queries are submitted to a reference monitor component that may be internal or external to the program. The use of a reference monitor does not guarantee that the program policy is correctly enforced, but a manual or semi-automated evaluation of the reference monitor interface is usually performed [17]. As an example, we consider the SELinux Policy Server [34]. A program that uses the

SELinux Policy Server, loads a *policy package* containing its policy into the SELinux Policy Server. The program is implemented with its own reference monitor interface which submits authorization queries to the Policy Server. We note that the programs that use an SELinux Policy Server may share labels, such as the labels of the system data, with other programs.

As an example, we previously reimplemented one of the trusted programs in an SELinux/MLS distribution, `logrotate`, using the Jif STL [13]. `logrotate` ages logs by writing them to new files and is trusted in SELinux/MLS because it can read and write logs of multiple MLS secrecy levels. Our experience from `logrotate` is that ensuring system security goals requires the trusted program to be aware of the system's label for its data. For example, if `logrotate` accesses a log file, it should control access to the file data based on the system (e.g., SELinux) label of that file. We manually designed the `logrotate` information flow policy to use the SELinux labels and the information flows that they imply. Further, since `logrotate` variables may also originate from program-specific data, such as configurations, in addition to system files, the information flow policy had to ensure that the information flows among system data and program data was also correct. As a result, the information flow policy required a manual merge of program and system information flow requirements.

## 2.2 Program Deployment

We must also consider how trusted programs are deployed on systems to determine what it takes to verify compliance. In Linux, programs are delivered in packages. A package is a set of files including the executable, libraries, configuration files, etc. A package provides new files that are specific to a program, but a program may also depend on files already installed in the system (e.g., system shared libraries, such as `libc`). Some packages may also export files that other packages depend on (e.g., special libraries and infrastructure files used by multiple programs).

For a trusted program, such as `logrotate`, we expect that a Linux package would include two additional, noteworthy files: (1) the program policy and (2) the SELinux policy module[2]. The program policy is the file that contains the declarative access control policy to be enforced by the program's reference monitor or STL implementation, as described above.

In SELinux, the system policy is now composed from the policy modules. SELinux policy modules specify the contribution of the package to the overall SELinux system policy [20]. While SELinux policy modules are specific to programs, they are currently designed by expert system administrators. Our `logrotate` program policy is derived from the program's SELinux policy module, and we envision that program policies and system policy modules will be designed in a coordinated way (e.g., by program developers rather than system administrators) in the future, although this is an open issue.

An SELinux policy module consists of three components that originate from three policy source files. First, a `.te` file defines a set of new SELinux types[3] for this package. It also defines the policy rules that govern program accesses to its own resources as well as system resources. Second, a `.fc` file specifies the assignment of package files to SELinux types. Some files may use types that are local to the policy module, but others may be assigned types defined previously (e.g., system types like `etc_t` is used for files in `/etc`). A `.if` file defines a set of interfaces that specify how other modules can access objects labeled with the types defined by this module.

When a package is installed, its files are downloaded onto the system and labeled based on the specification in the `.fc` file or the default system specification. Then, the trusted program's module policy is integrated into the SELinux system policy[4], enabling the trusted program to access system objects and other programs to access the trusted program's files. There are two ways that another program can access this package's files: (1) because a package file is labeled using an existing label or (2) another module is loaded that uses this module's interface or types. As both are possible for trusted programs, we must be concerned that the SELinux system policy may permit an untrusted program to modify a trusted program's package file.

For example, the `logrotate` package includes files for its executable, configuration file, documentation, `man` pages, execution status, etc. Some of these files are assigned new SELinux types defined by the `logrotate` policy module, such as the executable (`logrotate_exec_t`) and its status file (`logrotate_var_lib_t`), whereas others are assigned existing SELinux types, such as its configuration file (`etc_t`). The `logrotate` policy module uses system interfaces to obtain access to the system data (e.g., logs), but no other processes access `logrotate` interfaces. As a result, `logrotate` is only vulnerable to tampering because some of the system-labeled files that it provides may be modified by untrusted processes.

We are also concerned that a `logrotate` process may be tampered by the system data that it uses (e.g., Biba read-down [4]). For example, `logrotate` may read logs that contain malicious data. We believe that systems and programs should provide mechanisms to protect themselves from the system data that they process. Some interesting approaches have been proposed to protect process integrity [19, 30], so we consider this an orthogonal problem that we do not explore further here.

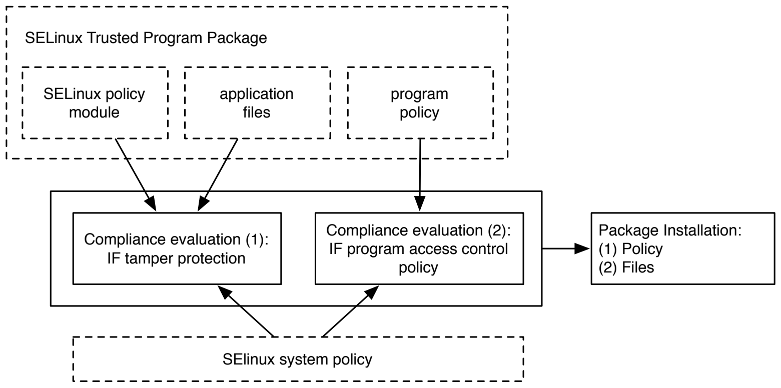

SELinux Trusted Program Package
SELinux policy module
application files
program policy
Compliance evaluation (1): IF tamper protection
Compliance evaluation (2): IF program access control policy
Package Installation: (1) Policy (2) Files
SElinux system policy

Figure 1: Deployment and Installation of a trusted package. First, we check two compliance goals: (1) the system protects the application and (2) the application enforces system goals. Second the package is installed: the policy module is integrated into the system policy and application files are installed.

## 2.3 Program Enforcement

To justify a system's trust, any trusted program must enforce a policy that complies with the system's security goals. The *reference monitor concept* [1] has been the guide for determining whether a system enforces its security goals, and we leverage this concept in defining compliance. A reference monitor requires three guarantees to be achieved: (1) *complete mediation* must ensure that all security-sensitive operations are authorized; (2) a reference monitor must be *tamperproof* to enforce its policy correctly; and (3) a reference monitor must be *simple enough to verify* enforcement of security goals. While the reference monitor concept is most identified with operating system security, a trusted program must also satisfy these guarantees to ensure that a system's security goal is enforced. As a result, we define that a program enforces a system's security goals if it satisfies the reference monitor guarantees in its deployment on that system.

In prior work, we developed a verification method that partially fulfilled these requirements. We developed a service, called SIESTA, that compares program policies against SELinux system policies, and only executes programs whose policies permit information flows authorized in the system policy [13]. This work considered two of the reference monitor guarantees. First, we used the SIESTA service to verify trust in the Jif STL implementation of the `logrotate` program. Since the Jif compiler guarantees enforcement of the associated program policy, this version of `logrotate` provides complete mediation, modulo the Java Virtual Machine. Second, SIESTA performs a policy analysis to ensure that the program policy complies with system security goals (i.e., the SELinux MLS policy). Compliance was defined as requiring that the `logrotate` policy only authorized an operation if the SELinux MLS policy) also permitted that operation. Thus, SIESTA is capable of verifying a program's enforcement of system security goals.

We find two limitations to this work. First, we had to construct the program access control policy relating system and program objects in an ad hoc manner. As the resultant program policy specified the union of the system and program policy requirements, it was much more complex than we envisioned. Not only is it difficult to design a compliant program access control policy, but that policy may only apply to a small number of target environments. As we discussed in Section 2.1, program policies should depend on system policies, particularly for trusted programs that we expect to enforce the system's policy, making them non-portable unless we are careful. Second, this view of compliance does not protect the trusted program from tampering. As described above, untrusted programs could obtain access to the trusted program's files after the package is installed, if the integrated SELinux system policy authorizes it. For example, if an untrusted program has write access to the `/etc` directory where configuration files are installed, as we demonstrated was possible in Section 2.2, SIESTA will not detect that such changes are possible.

In summary, Figure 1 shows that we aim to define an approach that ensures the following requirements:

- For any system deployment of a trusted program, automatically construct a program policy that is compliant with the system security goals, thus satisfying the reference monitor guarantee of being *simple enough to verify*.
- For any system deployment of a trusted program, verify, in a mostly-automated way, that the system policy does not permit tampering of the trusted program by any untrusted program, thus satisfying the reference monitor guarantee of being *tamperproof*. The typical number of verification errors must be

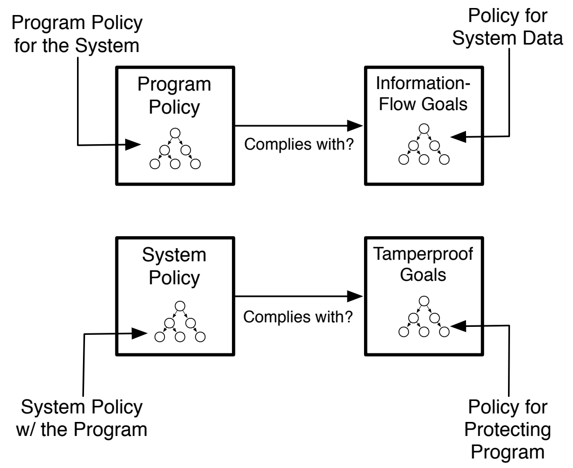


Figure 2: The two policy compliance problems: (1) verify that the *program policy* complies with the system's *information flow goals* and (2) verify that the *system policy*, including the program contribution (e.g., SELinux policy module), enforces the *tamperproofing goals* of the program.

small and there must be a set of manageable resolutions to any such errors.

In the remainder of the paper, we present a single approach that solves both of these requirements.

## 3 Policy Compliance

Verification of these two trusted program requirements results in the same conceptual problem, which we call *policy compliance problems*. Figure 2 shows these two problems. First, we must show that the program policy only authorizes operations permitted by the system's security goal policy. While we have shown a method by which such compliance can be tested previously [13,14], the program policy was customized manually for the system. Second, we also find that the system policy must comply with the program's tamperproof goals. That is, the system policy must not allow any operation that permits tampering the trusted program. As a result, we need to derive the tamperproof goals from the program (e.g., from the SELinux policy module).

In this section, we define the formal model for verifying policy compliance suitable for both the problems above. However, as can be seen from Figure 2, the challenge is to develop system security goals, program policies, and tamperproof goals in a mostly-automated fashion that will encourage successful compliance. The PIDSI approach in Section 4 provides such guidance.

### 3.1 Policy Compliance Model

We specify system-wide information-flow goals as a security lattice $\mathcal{L}$. We assume that elements of $\mathcal{L}$ have both an integrity and a confidentiality component: this is the case for both MLS labels in SELinux [11] and labels from the DLM [25]. Let $\mathsf{Integrity}(l)$ and $\mathsf{Conf}(l)$ be the integrity and confidentiality projections of a label $l \in \mathcal{L}$, respectively. Let the lattice $\mathcal{L}$ have both a top element, $\top$, and a bottom element $\perp$. We use $\mathsf{high} = \mathsf{Integrity}(\perp)$ and $\mathsf{low} = \mathsf{Integrity}(\top)$ to denote high and low integrity and write $\mathsf{high} \sqsubseteq \mathsf{low}$ to indicate that high integrity data can flow to a low integrity security label, but not the reverse.

An *information-flow graph* is a directed graph $G = (V, E)$ where $V$ is the set of vertices, each associated with a label from a security lattice $\mathcal{L}$. We write $V(G)$ for the vertices of $G$ and $E(G)$ for the edges of $G$, and for $v \in V(G)$ we write $\mathsf{Type}(v)$ for the label on the vertex $v$. Both subjects (e.g., processes and users) and objects (e.g., files and sockets) are assigned labels from the same security lattice $\mathcal{L}$. The edges in $G$ describe the information flows that a policy permits.

We now formally define the the concept of compliance between a graph $G$ and a security lattice $\mathcal{L}$. For $u, v \in V(G)$, we write $u \rightsquigarrow v$ if there is a path between vertices $u$ and $v$ in the graph $G$. An information-flow graph $G$ is compliant with a security lattice $\mathcal{L}$ if all paths through the combined information-flow graph imply that there is a flow in $\mathcal{L}$ between the types of the elements in the graph.

**Definition 3.1** (Policy Compliance)**.** An information-flow graph $G$ is *compliant* with a security lattice $\mathcal{L}$ if for each $u, v \in V(G)$ such that $u \rightsquigarrow v$, then $\mathsf{Type}(u) \sqsubseteq \mathsf{Type}(v)$ in the security lattice $\mathcal{L}$.

With respect to MAC policies, a positive result of the compliance test implies that the information-flow graph for a policy does not permit any operations that violate the information-flow goals as encoded in the lattice $\mathcal{L}$. If $G$ is the information flow graph of a trusted program together with the system policy, then a compliance test verifies that the trusted program only permits information flows allowed by the operating system, as we desire.

### 3.2 Difficulty of Compliance Testing

The main difficulty in compliance testing is in automatically constructing the program, system, and goal policies shown in Figure 2. Further, we prefer design constructions that will be likely to yield successful compliance.

The two particularly difficult cases are the *program policies* (i.e., upper left in the figure) and the *tamperproof goal policy* (i.e. lower right in the figure). The program policy and tamperproof goal policies require program requirements to be integrated with system requirements, whereas the system policy and system security goals are largely (although not necessarily completely) independent of the program policy. For example, while the system policy must include information flows for the program, the SELinux system includes policy modules for the `logrotate` and other trusted programs that can be combined directly.

First, it is necessary for *program policies* (i.e., upper left in the figure) to manage system objects, but often program policy and system policy are written with disjoint label sets. Thus, some mapping from program labels to system labels is necessary to construct a system-aware program policy before the information flow goals encoded in $\mathcal{L}$ can be evaluated. Let $P$ be an information-flow graph relating the program subjects and objects and and $S$ be information-flow graph relating the system subjects and objects. Let $P \oplus S$ be the policy that arises from combining $P$ and $S$ to form one information-flow graph through some sound combination operator $\oplus$; that is, if there is a runtime flow in the policy $S$ where the program $P$ has been deployed, then there is a flow in the information flow graph $P \oplus S$. Currently, there are no automatic ways to combine such program and system graphs into a system-aware program policy, meaning that $\oplus$ is implemented in a manual fashion. A manual mapping was used in previous work on compliance [13].

Second, the *tamperproof goal policy* (i.e., lower right in the figure) derives from the program's integrity requirements for its objects. Historically, such requirements are not explicitly specified, so it is unclear which program labels imply high integrity and which files should be assigned those high integrity labels. With the use of packages and program policy modules, the program files and labels are identified, but we still lack information about what defines tamperproofing for the program. Also, some program files may be created at installation time, rather than provided in packages, so the integrity of these files needs to be determined as well. We need a way to derive tamperproof goals automatically from packages and policy modules.

## 4 PIDSI Approach

We propose the *PIDSI approach* (**Program Integrity Dominates System Integrity**), where the trusted program objects (i.e., package files and files labeled using the labels defined by the module policy) are labeled such that their integrity is relative to all system objects. The information flows between the system and the trusted program can then be inferred from this relationship. We have found that almost all trusted program objects are higher integrity than system objects (i.e., system data should not flow to trusted program objects). One exception that we have found is that both trusted and untrusted programs are authorized to write to some log files. However, a trusted program should not depend on the data in a log file. While general cases may eventually be identified automatically as low integrity, at present we may have a small number of cases where the integrity level must be set manually.

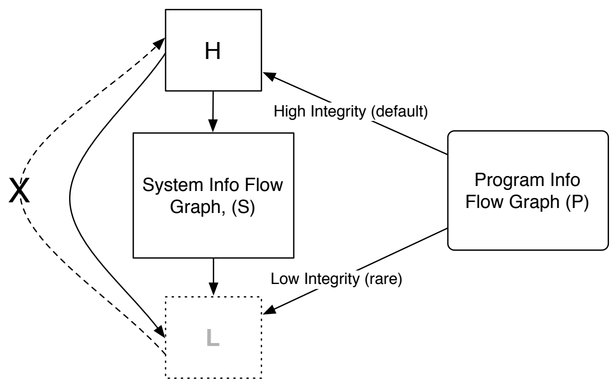


Figure 3: The PIDSI approach relates program labels $P$ to system labels $S$, such that the program-defined objects are higher integrity than the system data objects (assigned to $H$), with some small number of low integrity exceptions (assigned to $L$).

Our approach takes advantage of a distinction between the protection of the trusted program and protection of the data to which it is applied. Trusted program packages contain the files necessary to execute the program, and the integrity of the program's execution requires protection of these files. On the other hand, the program is typically applied to data whose protection requirements are defined by the system.

### 4.1 PIDSI Definition

By using the PIDSI approach between trusted program and the system, we can deploy that trusted program on different systems, ensuring compliance. Figure 3 demonstrates this approach. First, the program defines its own set of labels, which are designed either as high or low integrity. When the program is deployed, the system labels are placed in between the program's high and low integrity labels. This allows an easy check of whether a program is compliant with the system's policy, regardless of the specific mappings from system inputs and outputs to program inputs and outputs.

In the event that the trusted program allows data at a low integrity label to flow to a high label, then this approach can trick the system into trusting low integrity data. To eliminate this possibility, we automatically verify that no such flows are present in the program policy.

For confidentiality, we found that the data stored by most trusted programs was intended to be low secrecy. The only exception to this rule that we found in the trusted program core of SELinux was `sshd`; this program managed SSH keys at type `sshd_key_t`, which needed to be kept secret[5]. We note that if program data is low secrecy as well as high integrity the same information flows result, system data may not flow to program data, so no change is required to the PIDSI approach. Because of this, we primarily evaluate the PIDSI approach with respect to integrity.

In this context, the compliance problem requires checking that the system's policy, when added to the program, does not allow any new illegal flows. We construct the composed program policy $P'$ from $P$ and $S$. To composte $P$ and $S$ into $P' = P \oplus S$, first, split $P$ into subgraphs $H$ and $L$ as follows: if $u \in P$ is such that $\mathsf{Integrity}(\mathsf{Type}(u)) = \mathsf{high}$, then $u \in H$, and if $u \in P$ is such that $\mathsf{Integrity}(\mathsf{Type}(u)) = \mathsf{low}$, then $u \in L$. $P'$ contains copies of $S$, $H$, $L$, with edges from each vertex in $H$ to each vertex in $S$, and edges from each vertex in $S$ to $L$. The constructed system policy $P'$ corresponds to the deployment of the program policy $P$ on the system $S$.

**Theorem 4.1.** *Assume for all* $v \in V(P)$, $\mathsf{Conf}(\mathsf{Type}(v)) = \mathsf{Conf}(\perp)$. *Given test policy* $P$ *and target policy* $S$, *if for all* $u \in H$, $v \in L$, *there is no edge* $(v, u) \in P$, *then the test policy* $P$ *is compliant with the constructed system policy* $P'$.

Given the construction, the only illegal flow that can exist in $P'$ is from a vertex $v \in L$, which has a low integrity label, to one of the vertices $u \in H$, which has a high integrity label. The graph $S$ is compliant with $P'$ by definition, and the edges that we add between subgraphs are from $H$ to $S$ and $S$ to $L$: these do not upgrade integrity.

We argue that the PIDSI approach is consistent with the view of information flows in the trusted programs of classical security models. For example, MLS guards are trusted to downgrade the secrecy of data securely. Since an MLS guard must not lower the integrity of any downgraded data, it is reasonable to assume that the integrity of an MLS guard must exceed the system data that it processes. In the Clark-Wilson integrity model [7], only trusted *transformation procedures* (TPs) are permitted to modify high integrity data. In this model, TPs must be certified to perform such high integrity modifications securely. Thus, they also correspond to our notion of trusted programs. We find that other trusted programs, such as *assured pipelines* [5], also have a similar relationship to the data that they process.

## 4.2 PIDSI in Practice

In this section, we describe how we use the PIDSI approach to construct the two policy compliance problems defined in Section 3 for SELinux trusted programs. Our proposed mechanism for checking compliance of a trusted program during system deployment was presented in Figure 1: we now give the specifics how this procedure would work during an installation of

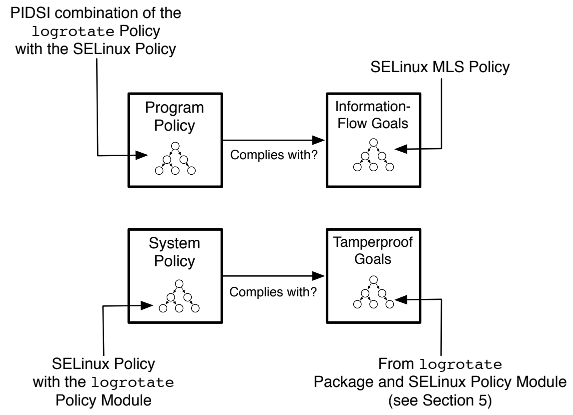

PIDSI combination of the logrotate Policy with the SELinux Policy
SELinux MLS Policy
Program Policy
Complies with?
Information-Flow Goals
System Policy
Complies with?
Tamperproof Goals
SELinux Policy with the logrotate Policy Module
From logrotate Package and SELinux Policy Module (see Section 5)

Figure 4: `logrotate` instantiation for the two policy compliance problems: (1) the *program policy* is derived using the PIDSI approach and the SELinux MLS policy forms the system's *information flow goals* and (2) the *system policy* is combined with the `logrotate` SELinux policy module and the *tamperproofing goals* are derived from the `logrotate` Linux package.

`logrotate`. Figure 4 shows how we construct both problems for `logrotate` on an SELinux/MLS system. For testing compliance against the system security goals, we use the PIDSI approach to construct the `logrotate` program policy and use the SELinux/MLS policy for the system security goals. For testing compliance against the tamperproof goals, we use the SELinux/MLS policy that includes the `logrotate` policy module for the system policy and we construct the tamperproof goal policy from the `logrotate` package. We argue why these constructions are satisfactory for deploying trusted programs, using `logrotate` on SELinux/MLS as an example.

For system security goal compliance, we must show that the program policy only permits information flows in the system security goal policy. We use the PIDSI approach to construct the program policy as described above. For the Jif version of `logrotate`, this entails collecting the types (labels) from its SELinux policy module, and composing a Jif policy lattice where these Jif version of these labels are higher integrity (and lower secrecy) than the system labels. Rather than adding each system label to the program policy, we use a single label as a template to represent all of the SELinux/MLS labels [13]. We use the SELinux/MLS policy for the security goal policy. This policy clearly represents the requirements of the system, and `logrotate` adds no additional system requirements. While some trusted programs may embody additional requirements that the system must uphold (e.g., for individual users), this is not the case for `logrotate`. As a result, to verify compliance we must show that there are no information flows in the program policy from system labels to program labels, a problem addressed by previous work [13].

For tamperproof goal compliance, we must show that the system policy only permits information flows that are authorized in the tamperproof goal policy. The system policy includes the `logrotate` policy module, as the combination defines the system information flows that impact the trusted program. The tamperproof policy is generated from the `logrotate` package and its SELinux policy module. The `logrotate` package identifies the labels of files used in the logrotate program. In addition to these labels, any new labels defined by the `logrotate` policy module, excepting process labels which are protected differently as described in Section 2.2, are also added to the tamperproof policy. The idea is that these labels may not be modified by untrusted programs. That is, untrusted process labels may not have any kind of write permission to the `logrotate` labels. Unlike security goal compliance, the practicality of tamperproof compliance is clear. It may be that system policies permit many subjects to modify program objects, thus making it impossible to achieve such compliance. Also, it may be difficult to correctly derive tamperproof goal policies automatically. In Section 5, we show precisely how we construct tamperproof policies and test compliance, and examine whether tamperproof compliance, as we have defined it here, is likely to be satisfied in practice.

## 5 Verifying Compliance in SELinux

In this section, we evaluate the PIDSI approach against actual trusted programs in the SELinux/MLS system. As we discussed in Section 4.2, we want to determine whether it is possible to automatically determine tamperproof goal policies and whether systems are likely to comply with such policies. First, we define a method for generating tamperproof goal policies automatically and show how compliance is evaluated for the `logrotate` program. Then, we examine whether eight other SELinux trusted programs meet satisfy tamperproof compliance as well. This group of programs was selected because: (1) they are considered MLS-trusted in SELinux and (2) these programs have Linux packages and SELinux policy modules. Our evaluation finds that there are only 3 classes of exceptions that result from our compliance checking for all of these evaluated packages. We identify straightforward resolutions for each of these exceptions. As a result, we find that the PIDSI approach appears promising for trusted programs in practice.

### 5.1 Tamperproof Compliance

To show how tamperproof compliance can be checked, we develop a method in detail for the `logrotate` program on a Linux 2.6 system with a SELinux/MLS strict reference policy. To implement compliance checking with the tamperproof goals, we construct representations of the system (SELinux/MLS) policy and the program's tamperproof goal policy. Recall from Section 3 that all the information flows in the system policy must be authorized by the tamperproof goal policy for the policy to comply.

#### 5.1.1 Build the Tamperproof Goal Policy

To build the tamperproof goal policy, we build an information-flow graph that relates the program labels to system labels according to the PIDSI approach. Building this graph consists of the following steps:

1. Find the high integrity program labels.
2. Identify the trusted system subjects.
3. Add information flow edges between the program labels, trusted subject labels, and remaining (untrusted) SELinux/MLS labels authorized by the PIDSI approach.

**Find the high integrity program labels.** This step entails collecting all the labels associated with the program's files, as these will all be high integrity per the PIDSI approach. These labels are a union of the package file labels determined by the file contexts (`.fc` file in the SELinux policy module and the system file context) and the newly-defined labels in the policy module itself. First, the `logrotate` package includes the files indicated in Table 1. This table presents lists a set of files, the label assigned to each, whether such label is a program label (i.e., defined by the program's policy module) or a system label, and the result of the tamperproof compliance check, described below. Second, some program files may be generated after the package is installed. These will be assigned new labels defined in the program policy module. An example of a `logrotate` label that will be assigned to a file that is not included in the package is `logrotate_lock_t`. In Section 6, we discuss other system files that a trusted program may depend upon.

**Identify trusted subjects.** Trusted subjects are SELinux subjects that are entrusted with write permissions to trusted programs. Based on our experience in analyzing SELinux/MLS, we identify the following seven trusted subjects: `dpkg_script_t, dpkg_t, portage_t, rpm_script_t, rpm_t, sysadm_t, prelink_t`. These labels represent package managers and system administrators; package managers and system administrators must be authorized to modify trusted programs. These subjects are also trusted by programs other than `logrotate`. We would want to control what code is permitted to run as these labels, but that is outside the scope of our current controls.

| File | SELinux Type | Policy | Writers | Exceptions |
|---|---|---|---|---|
| /etc/logrotate.conf | etc_t | system | 18 | integrity |
| /etc/logrotate.d | etc_t | system | 18 | integrity |
| /usr/sbin/logrotate | logrotate_exec_t | module | 8 | no |
| /usr/share/doc/logrotate/CHANGES | usr_t | system | 7 | no |
| /usr/share/man/logrotate.gz | man_t | system | 8 | no |
| /var/lib/logrotate.status | logrotate_var_lib_t | module | 8 | no |

Table 1: `logrotate` Compliance Test Case and Results: there are two exceptions, but they originate from the same system label `etc_t`.

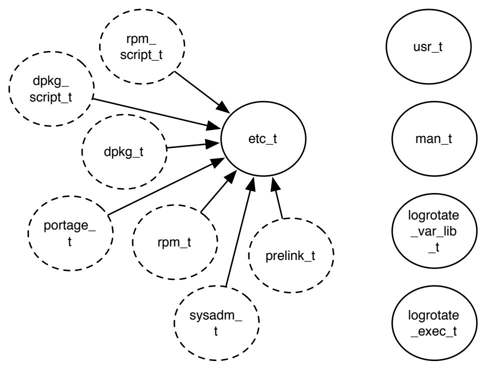


Figure 5: Part of the *tamperproof goal policy's* information-flow graph for `logrotate`. Only trusted labels (dotted line circles) and the program labels themselves are allowed to write to files with the program labels (solid line circles), which represent the high-integrity files according to the PIDSI approach. Not shown: edges from the trusted subjects to each of the program labels to the the right.

**Add information flow edges.** This step involves adding edges between vertices (labels) in the tamperproof goal information-flow graph based on the PIDSI approach. The PIDSI approach allows program labels to read and write each other, but the only SELinux/MLS labels that may write program labels are the trusted subjects (and read as well). Other SELinux labels are restricted to reading the program labels only. Figure 5 presents an example of a tamperproof goal policy's information-flow graph. Notice that only the system trusted labels (dotted circles) are allowed to write to program labels (solid line circles). The application has high integrity requirements for `etc_t`; the graph therefore includes edges that represent these requirements. The same set of edges are also added for the other program labels (presented to the right in the figure).

### 5.1.2 Build the System Policy

The system policy is represented as an information-flow graph (see Section 3). Building this graph consists of the following steps:

1. Create an information-flow graph that represents the current SELinux/MLS policy.
2. Add `logrotate` program's information flow vertices and edges based on its SELinux policy module.
3. Remove edges where neither vertex is in the tamperproof goal policy.

**Create an information flow graph.** We convert the current SELinux/MLS policy into an information-flow graph. Each of the labels in the SELinux/MLS policies is converted to a vertex. Information-flow edges are created by identifying *read-like* and *write-like* permissions [10, 29] for subject labels to objects labels. The following example illustrates the process we follow to create a small part of the graph. Rules 1-3 and 6 are system rules, rules 4-5 are module rules (defined in the `logrotate` policy module).

```
1.  allow init_t init_var_run_t:file
  {create getattr read append write
  setattr unlink};
2.  allow init_t bin_t:file
  {{read getattr lock execute ioctl}
  execute_no_trans};
3.  allow init_t etc_t:file
  {read getattr lock ioctl};
4.  allow logrotate_t etc_t:file
  {read getattr lock ioctl};
5.  allow logrotate_t bin_t:file
  {{read getattr lock execute ioctl}
  execute_no_trans};
6.  allow chfn_t etc_t:file
  {create ioctl read getattr write
  setattr append link unlink rename};
```

Figure 6 shows the result of the parsing of the previous rules. In this example, subjects with type `init_t` are allowed to read from and write to `init_var_run_t` and `logrotate_t` is allowed to read from `etc_t` and `bin_t`.

We note that Figure 6 shows that `chfn_t` has write access to `etc_t` which `logrotate_t` can read. While `logrotate` cannot write any file with the label `etc_t`,

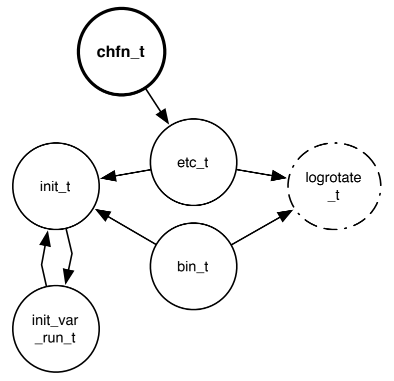


Figure 6: Information-flow graph for the system policy, including the `logrotate` program's policy module. `chfn_t` is not trusted to modify other trusted programs, but it has write access to `logrotate`'s files labeled `etc_t`.

it provides such a file via its package installation, so it depends on the integrity of files of the label. This will be identified as a tamperproof compliance exception below.

We are able to parse the text version of an SELinux policy (file `policy.conf`) with a C program integrated with Flex and Bison. We are also able to analyze the binary version of the SELinux system policy.

**Add `logrotate` program's information flows.** In a similar fashion to the method above, we extend the information flow graph with the vertices (labels) and edges (read and write flows) from the `logrotate` policy module.

**Remove edges where neither vertex is in the tamperproof goal policy.** As these flows cannot tamper the `logrotate` program, we remove these edges from the system policy for compliance testing.

### 5.1.3 Evaluating `logrotate`

This section presents how we automatically test tamperproof compliance. Tamperproof compliance is based checking the system policy for information flow integrity as defined by the tamperproof goal policy.

*Integrity Compliance Checking.* To detect integrity violations, we identify information flows that violate the Biba integrity requirement [4]: an information flow from a low integrity label (`type` in SELinux) to a high integrity label. `read` and `write` arguments are subject and object.

$$\begin{aligned}&NonBibaFlows_{SELinux}(Policy) = \\ &\quad \{(t_1,t_2) : t_1,t_2 \in types(Policy).\ highintegrity(t_1)\wedge \\ &\quad lowintegrity(t_2) \wedge (read(t_1,t_2) \vee write(t_2,t_1))\}\end{aligned}$$

We use the XSB Prolog engine [32] as the underlying platform. We developed a set of prolog queries based on the *NonBiba Flows* rule to detect the labels that affect compliance (i.e., the high integrity requirement that are not enforced by the system policy).

As mentioned in the previous section, we evaluate tamperproof compliance at installation time. Each time we load the policy graphs generated above into the Prolog engine and we run the integrity Prolog queries to determine if any flows satisfy (negatively) the *NonBiba Flows*, thus violating compliance.

**Results.** Table 1 presents the results for compliance checking `logrotate` against the generated tamperproof goal policy (see column 4). Only `etc_t` has unauthorized writers. In the SELinux/MLS reference policy, these writers are programs with legitimate reasons to write to files in the `/etc` directory, but none have legitimate reasons to write to `logrotate` files. For example, `chfn`, `groupdadd`, `passwd`, and `useradd` are programs that modify system files that store user information in `/etc`, `kudzu` is an program that detects and configures new and/or changed hardware in a system and requires to update its database stored in `/etc/sysconfig/hwconf`, and `updfstab` is designed to keep `/etc/fstab` consistent with the devices plugged in the system.

The obvious solution would be to refine the labels for files in `/etc` to eliminate these kinds of unnecessary and potentially-risky operations.

## 5.2 Evaluating other Trusted Programs

Table 2 shows a summary of the results from applying the PIDSI approach to eight SELinux trusted programs for which policy modules and packages are defined. The table shows: (1) trusted package, (2) file labels (SELinux types) used per package, (3) number of writers detected per type (Writers) and (4) exceptions. The integrity requirement assigned by default is high integrity for all types, except for the ones marked with **; because of the semantics associated to `/var`, various applications write to this directory, we assign low integrity requirement to `var_log_t` and `var_run_t`.

The common system types (`bin_t, etc_t, lib_t, man_t, sbin_t` and `usr_t`) are marked with † in the last two columns. The results for these types are displayed in Table 3. The results show only two exceptions, none in Table 2 and two in Table 3.

These reasons behind and resolutions for these exceptions are shown in Table 4. One good resolution would be a refinement of the policies: programs should have particular labels for their files, even if they are installed in system directories, instead of using general system labels. The use of a general system label gives all system

| Package | SELinux Label | Writers | Exception |
|---|---|---|---|
| cups | initrc_exec_t | 8 | no |
| | textrel_shlib_t | 9 | no |
| | lpr_exec_t | 8 | no |
| | dbusd_etc_t | 7 | no |
| | system types | † | † |
| | var_log_t** | 14 | no |
| | var_run_t** | 10 | no |
| | var_spool_t | 10 | no |
| dmidecode | dmidecode_exec_t | 8 | no |
| | system types | † | † |
| hald | locale_t | 7 | no |
| | initrc_exec_t | 8 | no |
| | hald_exec_t | 8 | no |
| | dbusd_etc_t | 7 | no |
| | system types | † | † |
| iptables | iptables_exec_t | 8 | no |
| | initrc_exec_t | 8 | no |
| | system types | † | † |
| kudzu | locale_t | 7 | no |
| | initrc_exec_t | 8 | no |
| | system types | † | † |
| Network Manager | initrc_exec_t | 8 | no |
| | NetworkManager_var_run_t | 8 | no |
| | NetworkManager_exec_t | 8 | no |
| | dbusd_etc_t | 7 | no |
| | system types | † | † |
| rpm | rpm_exec_t | 8 | no |
| | rpm_var_lib_t | 7 | no |
| | system types | † | † |
| | var_spool_t | 10 | no |
| sshd | sshd_exec_t | 8 | no |
| | sshd_var_run_t | 8 | no |
| | ssh_keygen_exec_t | 8 | no |
| | ssh_keysign_exec_t | 8 | no |

Table 2: Results of applying the PIDSI approach to SELinux Trusted Packages. Columns with a '†' are displayed in table 3

programs access to these files (case APP LABELS in Table 4). However, this option is not always possible, as sometimes a program actually requires access to system files. In such cases, the programs have to be trusted (case ADD in Table 4). For example, some trusted programs read information from the `/etc/passwd` file, so those subjects permitted to modify that file must be trusted. Only a small number of such programs must be trusted.

## 6 Discussion

Trusted programs may use system files, such as system libraries or the password file, in addition to the files provided in their packages. Because some of our trusted program packages installed their own libraries under the system label `lib_t` our analysis included system libraries. Therefore, application integrity not only depends on the integrity of the files in the installation package but also on some other files. In general, the files that the program execution depends on should be comprehensively identified. These should be well-known per system.

| SELinux Label | Writers | Exceptions |
|---|---|---|
| bin_t | 9 | no |
| etc_t | 18 | integrity |
| lib_t | 8 | no |
| man_t | 8 | integrity |
| sbin_t | 8 | no |
| usr_t | 7 | no |

Table 3: System labels referenced by the packages presented in Table 2. Only `etc_t` and `man_t` have conflicts; the number of conflicting types per case can not be high (Writers column is an upper limit since it includes trusted writers), so we can precisely examine each exception and suggest resolutions (shown in Table 4).

An issue is whether a trusted program may create a file whose integrity it depends upon that has a system label. For example, a trusted program generates the password file, but this used by the system, so it has a system label. We did not see a case where this happened for our trusted programs, but we believe that this is possible in practice. We believe that more information about the integrity of the contents generated by the program will need to be used in compliance testing. For example, if the program generates data it marks as high integrity, then we could leverage this in addition to package files and program policy labels to generate tamperproof goal policies.

An issue with our approach is the handling of low integrity program objects. Since low integrity program objects are the lowest integrity objects in the system, any program can write to these objects. We find that we want low integrity program objects to be relative to the trusted programs; lower than all trusted programs, but still higher than system data. Further investigation is required.

The approach in this paper applies only to trusted programs. We make no assumptions about the relationship between untrusted program and the system data. In fact, we are certain that there is system data that should not be accessed by most, if not all, untrusted programs. Note that there is no advantage to verifying the compliance of untrusted program, because the system does not depend on untrusted programs to enforce its security goals. Such programs have no special authority.

## 7 Related Work

*Policy Analysis.* Policies generally contain a considerable number of rules that express how elements in a given

| SELinux Label | Conflicting Labels | Type of Exception | Resolution Method | Comment |
|---|---|---|---|---|
| etc_t | groupadd_t, passwd_t, useradd_t, chfn_t | Integrity | ADD | The conflicting labels require access to the the same file `/etc/passwd` |
| etc_t | updfstab_t, ricci_modstorage_t, firstboot_t | Integrity | ADD | The first two labels have legitimate reasons to modify `/etc/fstab`. The last type modifies multiple files in `/etc` |
| etc_t | postgresql_t,kudzu_t | Integrity | APP LABELS | The conflicting types need access to application files labeled with system labels |
| man_t | system_crond_t | Integrity | REMOVE | crond does not need to write manual pages |
| ADD: Add conflicting types to the set of trusted readers (confidentiality) or writers (integrity). APP LABELS: The associated application requires access to a file that is application specific but was labeled using system labels. Adding application specific labels to handle those files solves the conflict. REMOVE: The permission requested is not required | | | | |

Table 4: **Compliance Exceptions and Resolutions**. This table details the exceptions to tamperproof compliance presented in Table 3. It shows the list of conflicting, untrusted subjects and the resolution method, per case.

environment must be controlled. Because of the size of a policy and the relationships that emerge from having a large number of rules, it is difficult to manually evaluate whether a policy satisfies a given property or not. As a consequence, tools to automatically analyze policy are necessary. APOL [35], PAL [29], SLAT [10], Gokyo [16] and PALMS [15] are some of the tools developed to analyze SELinux policies; however, each of these tools focuses on the analysis of single security policies. Of these, only PALMS offers mechanisms to compare policies; in particular it addresses compliance evaluation, but our approach to compliance is broader and allows the compliance problem to be automated.

*Policy Modeling*. We need a formal model to reason about the features of a given policy. Such a model should be largely independent of particular representation of the targeted policies and should enable comparisons among different policies. Multiple models have been proposed and each one of them defines a set of components that need to be considered when translating a policy to an intermediate representation. Cholvy and Cuppens [6] focus on permissions, obligations, prohibitions and provide a mechanism to check regulation consistency. Bertino et al. [3] focus on subjects, objects and privileges, as well as the organization of these components and the set of authorization rules that define the relationships among components and the set of derived rules that may be generated because of a hierarchical organization. Kock et al. [18] represent policies as graphs with nodes that represent components(processes, users, objects) and edges that represent rules and a set of constraints that globally applied to the system. In any case, policy modeling becomes a building block in the process of evaluating compliance. Different policies must be translated to an intermediate representation (a common model) so they can be compared and their properties evaluated.

*Policy Reconciliation*. Policy compliance problems may resemble policy reconciliation problems. Given two policies A and B that define a set of requirements, a reconciliation algorithm looks for a specific policy instance C that satisfies the stated requirements. Policy compliance in a general sense, i.e. 'Given a policy A and a policy B, is B compliant with A?' means 'is any part of A in conflict with B?'. Previous work [21] shows that reconciliation of three or more policies is intractable. Compliance is also a intractable problem since this would require to checking all possible paths in B against all possible paths in A. Although both of these problems are similar in that they both test policy properties and are non-tractable in general cases (no restrictions), they differ in their inputs and expected outputs. While in the case of reconciliation, an instance that satisfies the requirements has to be calculated, in the case of compliance, policy instances are given and one is evaluated against the other one.

*Policy Compliance*. The security-by-contract paradigm resembles our policy compliance model. It is one of the mechanisms proposed to support installation and execution of potentially malicious code from a third party in a local platform. Third party applications are expected to come with a security contract that specifies application behavior regarding security issues. The first step in the verification process is checking whether the behaviors allowed by the contract are also allowed by the local policy [8]. In the most recent project involving contract matching, contract and policy are security automatons and the problem of contract matching becomes a problem of testing language inclusion for automatons. While there is no known polynomial technique to test language inclusion for non-deterministic automatons, determining language inclusion for deterministic automatons is known to be polynomial [9]. One main advantage of our representation is that we are verifying policies that are actually implemented by the enforcing mechanism, not high level statements that may not be actually implemented because of the semantic gap between specification and implementation. In addition, the enforcing mechanism is part of the architecture.

## 8 Conclusion

This work is driven by the idea of unifying application and system security policies. Since applications and systems policies are independently developed, they use different language syntax and semantics. As a consequence, it is difficult to prove or disprove that programs enforce system security goals. The emergence of mandatory access control systems and security typed languages makes it possible to automatically evaluate whether applications and systems enforce common security goals. We reshape this problem as a verification problem: we want to evaluate if applications are *compliant* with system policies.

We found that compliance verification involves two tasks: we must ensure that the system protects application from being tampered with, as well as verify that the application enforces system security goals. In order to automate the mapping between the program policy and the system policy, we proposed the PIDSI (**Program Integrity Dominates System Data Integrity**) approach. The PIDSI approach relies on the observation that in general program objects are higher integrity than system objects. We tested the trusted program core of the SELinux system to see if its policy was compatible with the PIDSI approach. We found that our approach accurately represents the SELinux security design with a few minor exceptions, and requires little or no feedback from administrators in order to work.

## Notes

[1]The program verification (e.g., STL compilation) enforces the *complete mediation* guarantee.

[2]At present, module policies are not included in Linux packages, but RedHat, in particular, is interested in including SELinux module policies in its `rpm` packages in the future [36].

[3]SELinux uses the term *type* for its labels, as it uses an extended Type Enforcement policy [5].

[4]As described above, this must be done manually now, via `semodule`, but the intent is that when you load a package containing a module policy, someone will install the module policy.

[5]In this case, violating the confidentiality of SSH keys enables a large class of integrity attacks. This phenomenon has been discussed more generally by Sean Smith [31].

## References

[1] ANDERSON, J. P. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1972.

[2] BELL, D. E., AND LAPADULA, L. J. Secure computer systems: Unified exposition and multics interpretation. Tech. rep., MITRE MTR-2997, March 1976.

[3] BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. A logical framework for reasoning about access control models. In *Proceedings of SACMAT* (2001).

[4] BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[5] BOEBERT, W. E., AND KAIN, R. Y. A practical alternative to heirarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (1985).

[6] CHOLVY, L., AND CUPPENS, F. Analyzing Consistency of Security Policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1997), pp. 103–112.

[7] CLARK, D. D., AND WILSON, D. A comparison of military and commercial security policies. In *1987 IEEE Symposium on Security and Privacy* (May 1987).

[8] DESMET, L., JOOSEN, W., MASSACCI, F., NALIUKA, K., PHILIPPAERTS, P., PIESSENS, F., AND VANOVERBERGHE, D. A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of the ACM Computer Security Architecture Workshop* (2007).

[9] DRAGONI, N., MASSACCI, F., NALIUKA, K., SEBASTIANI, R., SIAHAAN, I., QUILLIAN, T., MATTEUCCI, I., AND SHAEFER, C. Methodologies and tools for contract matching. Security of Software and Services for Mobile Systems.

[10] GUTTMAN, J. D., HERZOG, A. L., RAMSDELL, J. D., AND SKORUPKA, C. W. Verifying information flow goals in Security-Enhanced Linux. *J. Comput. Secur. 13*, 1 (2005), 115–134.

[11] HANSON, C. SELinux and MLS: Putting the pieces together. In *Proceedings of the 2nd Annual SELinux Symposium* (2006).

[12] HICKS, B., KING, D., MCDANIEL, P., AND HICKS, M. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)* (Ottawa, Canada, June 10 2006), ACM Press.

[13] HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[14] HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. Integrating SELinux with security-typed languages. In *Proceedings of the 3rd SELinux Symposium* (Baltimore, MD, USA, March 2007).

[15] HICKS, B., RUEDA, S., ST. CLAIR, L., JAEGER, T., AND MCDANIEL, P. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)* (Antipolis, France, June 2007).

[16] JAEGER, T., EDWARDS, A., AND ZHANG, X. Managing access control policies using access control spaces. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies* (2002), ACM Press, pp. 3–12.

[17] JAEGER, T., EDWARDS, A., AND ZHANG, X. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security (TISSEC) 7*, 2 (May 2004), 175–205.

[18] KOCK, M., MACINI, L., AND PARISI-PRESICCE, F. On the specification and evolution of access control policies. In *Proceedings of SACMAT* (2001).

[19] LI, N., MAO, Z., AND CHEN, H. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy* (2007).

[20] MAYER, F., MACMILLAN, K., AND CAPLAN, D. *SELinux by Example*. Prentice Hall, 2007.

[21] MCDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. *ACM Transactions on Information and System Security V*, N (May 2006), 1–32.

[22] McGraw, G., and Felten, E. *Java Security*. Wiley Computer, 1997.

[23] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL '99*, pp. 228–241.

[24] Myers, A. C., and Liskov, B. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (October 1997).

[25] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[26] Myers, A. C., Nystrom, N., Zheng, L., and Zdancewic, S. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[27] Security-enhanced Linux. Available at `http://www.nsa.gov/selinux`.

[28] Pottier, F., and Simonet, V. Information Flow Inference for ML. In *Proceedings ACM Symposium on Principles of Programming Languages* (Jan. 2002), pp. 319–330.

[29] Sarna-Starosta, B., and Stoller, S. Policy analysis for Security-Enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)* (April 2004), pp. 1–12. Available at http://www.cs.sunysb.edu/˜stoller/WITS2004.html.

[30] Shankar, U., Jaeger, T., and Sailer, R. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (NDSS'06)* (San Diego, CA, USA, Feb. 2006).

[31] Smith, S. W. Outbound authentication for programmable secure coprocessors. In *European Symposium on Research in Computer Security (ESORICS)* (2002), pp. 72–89.

[32] Stony Brook University. Computer Science Department. XSB: Logic programming and deductive database system for Unix and Windows. Available at `http://xsb.sourceforge.net`.

[33] Swamy, N., Corcoran, B., and Hicks, M. Fable: A language for enforcing user-defined security policies. In *In Proceedings of the IEEE Symposium on Security and Privacy (Oakland), May 2008. To appear.*

[34] Tresys Technology. SELinux Policy Server. Available at http://www.tresys.com/selinux/selinux_policy_server.

[35] Tresys Technology. SETools - policy analysis tools for SELinux. available at http://oss.tresys.com/projects/setools.

[36] Walsh, D. SELinux Mailing List. `http://www.engardelinux.org/modules/index/list_archives.cgi?list=selinu%x&page=0609.html&month=2007-12`, December 2007.

[37] The X Foundation: `http://www.x.org`.

# Helios: Web-based Open-Audit Voting

Ben Adida
ben_adida@harvard.edu
*Harvard University*

## Abstract

Voting with cryptographic auditing, sometimes called open-audit voting, has remained, for the most part, a theoretical endeavor. In spite of dozens of fascinating protocols and recent ground-breaking advances in the field, there exist only a handful of specialized implementations that few people have experienced directly. As a result, the benefits of cryptographically audited elections have remained elusive.

We present Helios, the first web-based, open-audit voting system. Helios is publicly accessible today: anyone can create and run an election, and any willing observer can audit the entire process. Helios is ideal for online software communities, local clubs, student government, and other environments where trustworthy, secret-ballot elections are required but coercion is not a serious concern. With Helios, we hope to expose many to the power of open-audit elections.

## 1 Introduction

Over the last 25 years, cryptographers have developed election protocols that promise a radical paradigm shift: election results can be verified entirely by public observers, all the while preserving voter secrecy. These protocols are said to provide two properties: *ballot casting assurance*, where each voter gains personal assurance that their vote was correctly captured, and *universal verifiability*, where any observer can verify that all captured votes were properly tallied. Some have used the term “open-audit elections” to indicate that anyone, even a public observer with no special role in the election, can act as auditor.

Unfortunately, there is a significant public-awareness gap: few understand that these techniques represent a fundamental improvement in how elections can be audited. Even voting experts who recognize that open-audit elections are “the way we’ll all vote in the future” seem to envision a *distant* future, not one we should consider for practical purposes yet. The few implementations of open-audit elections that do exist [3, 2] have not had as much of an impact as hoped, in large part because they require special equipment and an in-person experience, thus limiting their reach.

We present Helios, a web-based open-audit voting system. Using a modern web browser, anyone can set up an election, invite voters to cast a secret ballot, compute a tally, and generate a validity proof for the entire process. Helios is deliberately simpler than most complete cryptographic voting protocols in order to focus on the central property of *public auditability*: any group can outsource its election to Helios, yet, even if Helios is fully corrupt, the integrity of the election can be verified.

**Low-Coercion Elections.** Voting online or by mail is typically insecure in high-stakes elections because of the coercion risk: a voter can be unduly influenced by an attacker looking over her shoulder. Some protocols [13] attempt to reduce the risk of coercion by letting voters override their coerced vote at a later (or earlier) time. In these schemes, the privacy burden is shifted from vote casting to voter registration. In other words, no matter what, *some truly private interaction* is required for coercion resistance.

With Helios, we do not attempt to solve the coercion problem. Rather, we posit that a number of settings—student government, local clubs, online groups such as open-source software communities, and others—do not suffer from nearly the same coercion risk as high-stakes government elections. Yet these groups still need voter secrecy and trustworthy election results, properties they cannot currently achieve short of an in-person, physically observable and well orchestrated election, which is often not a possibility. We produced Helios for exactly these groups with low-coercion elections.

**Trust no one for integrity, trust Helios for privacy.** In cryptographic voting protocols, there is an inevitable compromise: unconditional integrity, or unconditional privacy. When every component is compromised, only one of those two properties can be preserved. In this work, we hold the opinion that the more important property, the one that gets people's attention when they understand open-audit voting, is unconditional integrity: even if all election administrators are corrupt, they cannot convincingly fake a tally. With this design decision made, privacy is then ensured by recruiting enough trustees and hoping that a minimal subset of them will remain honest.

In the spirit of simplicity, and because it is difficult to explain to users how privacy derives from the acts of multiple trustees, Helios takes an interesting approach: there is only one trustee, the Helios server itself. Privacy is guaranteed only if you trust Helios. Integrity, of course, does not depend on trusting Helios: the election results can be fully audited even if all administrators – in this case the single Helios sever – is corrupt. Future versions of Helios may support multiple trustees. However, exhibiting the power of universal verifiability can be achieved with this simpler setup.

**Our Contribution.** In this work, we contribute the software design and an open-source, web-based implementation of Helios, as well as a running web site that anyone can use to manage their elections at `http://heliosvoting.org`. We do not claim any cryptographic novelty. Rather, our contribution is a combination of existing Web programming techniques and cryptographic voting protocols to provide the first truly accessible open-audit voting experience. We believe Helios provides a unique opportunity to educate people about the value of cryptographic auditability.

**Limitations.** While every major feature is functional, Helios is currently alpha software. As such, it requires Firefox 2 (or later). In addition, some aspects of the user interface, especially for administrative tasks, require significant additional polish and better user feedback on error. These issues are being actively addressed.

**This Paper.** In Section 2, we briefly review the Helios protocol, based on the Benaloh vote-casting approach [5] and the Sako-Kilian mixnet [16]. In Section 3, we cover some interesting techniques used to implement Helios in a modern Web browser. Section 4 covers the specifics of the Helios system and its use cases. We discuss, in Section 5, the security model, some performance metrics, and features under development. We reference related work in Section 6 and conclude in Section 7.

## 2 Helios Protocol

This section describes the Helios protocol, which is most closely related to Benaloh's Simple Verifiable Voting protocol [5], which itself is partially inspired by the Sako-Kilian mixnet [16]. We claim no novelty, we only mean to be precise in the steps taken by voters, administrators, and auditors, and we mean to provide enough details for an able programmer to re-implement every portion of this protocol.

### 2.1 Vote Preparation & Casting

The key auditability feature proposed by Benaloh's Simple Verifiable Voting is the separation of ballot preparation and casting. A ballot for an election can be viewed and filled in by anyone at any time, without authentication. The voter is authenticated only at *ballot casting* time. This openness makes for increased auditability, since *anyone*, including an auditor not eligible to vote (e.g. someone from a political organization who has already voted), can test the ballot preparation mechanism. The process is as follows between Alice, the voter, and the Ballot Preparation System (BPS):

1. Alice begins the voting process by indicating in which election she wishes to participate.
2. The BPS leads Alice through all ballot questions, recording her answers.
3. Once Alice has confirmed her choices, the BPS encrypts her choices and commits to this encryption by displaying a hash of the ciphertext.
4. Alice can now choose to *audit* this ballot. The BPS displays the ciphertext and the randomness used to create it, so that Alice can verify that the BPS had correctly encrypted her choices. If this option is selected, the BPS then prompts Alice to generate a new encryption of her choices.
5. Alternatively, Alice can choose to *seal* her ballot. The BPS discards all randomness and plaintext information, leaving only the ciphertext, ready for casting.
6. Alice is then prompted to authenticate. If successful, the encrypted vote, which the BPS committed to earlier, is recorded as Alice's vote.

Because we worry little about the possibility of coercion, Helios can be simpler than the Benaloh system that inspired it. Specifically, the BPS does not sign the ciphertext before casting, and we do not worry about Alice seeing the actual hash commitment of her encrypted vote before sealing. (The subtle reasons why these can lead to coercion are explained in [6].)

**Teaching Voters about Coercion.** While online-only voting is inherently coercible, few voters are aware of this issue: many US elections today are shifting to vote-by-mail without realizing the subtle but critical change in coercibility. We take the opportunity to make this issue clear with Helios by making coercion explicit: we provide a "Coerce Me!" button at ballot casting time which allows any voter to email a potential coercer the complete proof – ciphertext, randomness, and plaintext – of how they voted. This design choice does not enable new avenues for coercion; it only makes the existing coercibility more apparent. It is our hope that Helios can thus help educate voters about this critical election issue.

## 2.2 Bulletin Board of Votes

In all cryptographic voting protocols, a bulletin board is made publicly available. On this bulletin board, cast votes are displayed next to either a voter name or voter identification number. All subsequent data processing is also posted for the public to download and verify. A number of distributed bulletin board protocols, including consensus algorithms, have been proposed.

In Helios, we forgo complexity and opt for the simplest possible bulletin board, run by a single server. We expect auditors to check the bulletin board's integrity over time, and enough individual voters to check that their encrypted vote appears on the bulletin board. Once again, we opt for this simplification in order to focus the user on the major advantage of the system: Helios is auditable by anyone, including watchdog organizations and individual voters themselves.

## 2.3 Sako-Kilian/Benaloh Mixnet

In cryptographic voting protocols that wish to preserve individual ballots and potentially support write-in votes, anonymization is typically achieved by way of a *mixnet*, where trustees each shuffle and re-randomize the cast ciphertexts before jointly decrypting them. Both the shuffling and decryption of encrypted ballots are accompanied by proofs of correctness.

We use the Sako-Kilian protocol [16], the first provable mixnet based on El-Gamal re-encryption. We note that Benaloh uses a very similar technique [5]. We chose this scheme because of its simplicity and ease of explanation, even though we know of more complex protocols [15] that achieve an order of magnitude better performance for the same assurance of integrity.

**El Gamal.** Recall the El-Gamal encryption scheme implemented to support semantic security: a large (1024 bits) prime $p$ is selected, such that $p = 2q + 1$ with $q$ also prime. A generator $g$ of the $q$-order subgroup of $\mathsf{Z}_p^*$ is selected. A secret key $x \in \mathsf{Z}_q$ is selected, and the corresponding public key $y = g^x \bmod p$ is computed. A message $m$ in the $q$-order subgroup of $\mathsf{Z}_p^*$ is then encrypted by selecting $r \in \mathsf{Z}_q$ and computing: $c = (\alpha, \beta) = (g^r, my^r)$. Decryption is computed as $m = \alpha^{-x}\beta$.

When $m \in \mathsf{Z}_q$, meaning that it is not necessarily in the $q$-order subgroup of $\mathsf{Z}_p^*$, a simple mapping from $\mathsf{Z}_q$ to the $q$-order subgroup of $\mathsf{Z}_p^*$ is used: on input $m$, compute $m_0 = m + 1$ and, if $m_0^q \equiv 1 \bmod p$, output $m_0$, otherwise output $-m_0 \bmod p$. Upon decryption, one obtains $m$, and the reverse mapping is achieved as follows: if $m \leq q$, set $m_0 = m$, otherwise $m_0 = -m \bmod p$, and output $m_0 - 1$. Using these techniques, we can efficiently encrypt and decrypt messages in $\mathsf{Z}_q$ for $q$ a 512-bit prime. This is the natural path for message encryption, as a typical plaintext can be any string of bits up to a certain size.

**Re-encryption.** The El-Gamal cryptosystem offers simple re-encryption, even when using the $\mathsf{Z}_q$ mapping given above. Given a ciphertext $c = (\alpha, \beta)$, a ciphertext $c'$ can be computed by selecting $s \in \mathsf{Z}_q$ and computing $c' = (g^s\alpha, y^s\beta)$. It is clear that $c'$ and $c$ decrypt to the same plaintext, $c$ with randomness $r$ and $c'$ with randomness $r + s$.

**Sako-Kilian Shuffle & Proof.** In the Sako-Kilian mixnet, all inputs are El-Gamal ciphertexts. A mix server takes $N$ inputs, re-encrypts them using re-encryption factors $\{s_i\}_{i\in[1,N]}$ and permutes them according to random permutation $\pi_N$, so that $d_i = \mathsf{Reenc}(c_{\pi(i)}, s_i)$.

To prove that it mixed its inputs correctly, a mix server produces a second, "shadow mix," as illustrated in Figure 1. The verifier then challenges the mix server to reveal the permutation and re-encryption factors for either this shadow mix or the difference between the two mixes, i.e. the shuffle that would transform the shadow mix outputs into the primary mix outputs. An honest mix server can obviously answer either challenge, while a cheating mix server can answer at most one of those questions convincingly, and is thus caught with at least 50% probability. To increase the assurance of integrity, we ask the mix server to produce a few shadow mixes. The verifier then provides the appropriate number of challenge bits, one for each shadow mix. If the mix server succeeds at responding to all challenges, then the primary mix is correct with probability $1 - 2^{-t}$, where $t$ is the number of shadow mixes. Choosing $t = 80$ guarantees integrity with overwhelming probability.

In Helios, we need a non-interactive proof: there are many verifiers and we do not wish to perform such heavy computation for everyone who requests it. The proof protocol described above, which is Honest-Verifier Zero-Knowledge (HVZK), is thus transformed using the Fiat-

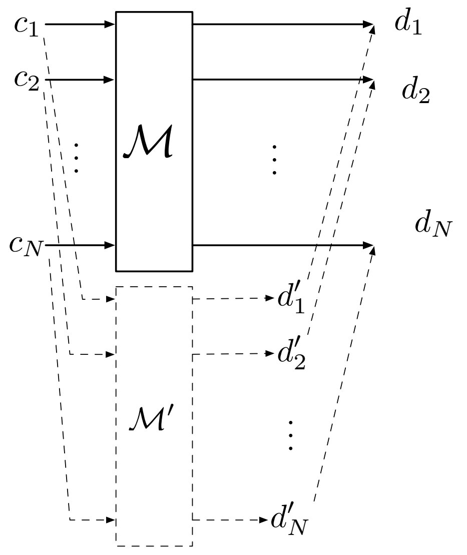


Figure 1: The "Shadow-Mix" Shuffle Proof. The mix server creates a secondary mix. If challenged with bit 0, it reveals this secondary mix. If challenged with bit 1, it reveals the "difference" between the two mixes.

Shamir heuristic [9]: the challenge bits are computed as the hash of all shadow mixes. Note how this approach is only workable if we have enough shadow mixes to provide an overwhelming probability of integrity: if there is a non-negligible probability of cheating, a cheating prover can produce many shadow mixes until it finds a set whose hash provides just the right challenge bits to cheat.

**Proof of Decryption.** Once an El Gamal ciphertext is decrypted, this decryption can be proven using the Chaum-Pedersen protocol [8] for proving discrete-logarithm equality. Specifically, given a ciphertext $c = (\alpha, \beta)$ and claimed plaintext $m$, the prover shows that $\log_g(y) = log_\alpha(\beta/m)$:

- The prover selects $w \in \mathsf{Z}_q$ and sends $A = g^w, B = \alpha^w$ to the verifier.
- The verifier challenges with $c \in \mathsf{Z}_q$.
- The prover responds with $t = w + xc$.
- The verifier checks that $g^t = Ay^c$ and $\alpha^t = B(\beta/m)^c$.

It is clear that, given $c$ and $t$, $A$ and $B$ can be easily computed, thus providing for simulated transcripts of such proofs indicating Honest-Verifer Zero-Knowledge. It is also clear that, if one could rewind the protocol and obtains prover responses for two challenge values against the same $A$ and $B$, the value of $x$ would be easily solvable, thus indicating that this is a proof of knowledge of the discrete log and that $\log_g(y) = log_\alpha(\beta/m)$.

As this protocol is HVZK with overwhelming probability of catching a cheating prover, it can be transformed safely into non-interactive form using the Fiat-Shamir heuristic. We do exactly this in Helios to provide for non-interactive proofs of decryption that can be posted publicly and re-distributed by observers.

### 2.4 The Whole Process

The entire Helios protocol thus unfolds as follows:

1. Alice prepares and audits as many ballots as she wishes, ensuring that all of the audited ballots are consistent. When she is satisfied, Alice casts an encrypted ballot, which requires her to authenticate.
2. The Helios bulletin board posts Alice's name and encrypted ballot. Anyone, including Alice, can check the bulletin board and find her encrypted vote posted.
3. When the election closes, Helios shuffles all encrypted ballots and produces a non-interactive proof of correct shuffling, correct with overwhelming probability.
4. After a reasonable complaint period to let auditors check the shuffling, Helios decrypts all shuffled ballots, provides a decryption proof for each, and performs a tally.
5. An auditor can download the entire election data and verify the shuffle, decryptions, and tally.

If an election is made up of more than one race, then each race is treated as a separate election: each with its own bulletin board, its own independent shuffle and shuffle proof, and its own decryptions. This serves to limit the possibility of re-identifying voters given long ballots where any given set of answers may be unique in the set of cast ballots.

## 3 Web Components

We have clearly stated that Helios values integrity first, and voter privacy second. That said, Helios still takes great care to ensure voter privacy, using a combination of modern Web programming techniques. Once the ballot is loaded into the browser, all candidate selections are recorded within the browser's memory, without any further network calls until the ballot is encrypted and the plaintext is discarded. In this section, we cover the Web components we use to accomplish this goal.

## 3.1 Single-Page Web Application

A number of Web applications today are called “single-page applications” in that the page context and its URL never change. Gmail [10] is a prime example: clicks cause background actions rather than full-page loads. The technique behind this type of Web application is the use of JavaScript to handle user clicks:

```
<a onclick="do_stuff()" href="#">Do Stuff</a>
```

When a user clicks the “Do Stuff” link, no new page is loaded. Instead, the JavaScript function `do_stuff()` is invoked. This function may make network requests and update the page’s HTML, but, importantly, the page context, including its JavaScript scope, is preserved.

For our purposes, the key point is that, if all necessary data is pre-loaded, the `do_stuff()` function may not need to make any network calls. It can update some of its scope, read some of its pre-loaded data, and update the rendered HTML user interface accordingly. This is precisely the approach we use for our ballot preparation system: the browser loads all election parameters, then leads the voter through the ballot without making any additional network requests.

**The jQuery JavaScript Library.** Because we expect auditors to take a close look at our browser-based JavaScript code, it is of crucial importance to make this code as concise and legible as possible. For this purpose, we use the jQuery JavaScript library, which provides flexible constructs for accessing and updating portions of the HTML Document Object Model (DOM) tree, manipulating JavaScript data structures, and making asynchronous network requests (i.e. AJAX). An auditor is then free to compare the hash of the jQuery library we distribute with that of the official distribution from the jQuery web site.

**JavaScript-based Templating.** Also important to the clarity of our browser-based code is the level of intermixing of logic and presentation: when all logic is implemented in JavaScript, it is tempting to intermix small bits of HTML, which makes for code that is particularly difficult to follow. Instead, we use the jQuery JavaScript Templating library. Then, we can bind a template to a portion of the page as follows:

```
$("#main").setTemplateURL(
  "/templates/election.html"
);
```

which connects to an HTML template with variable placeholders:

```
<p>
  The election hash is {$T.election.hash}.
</p>
```

The code can, at a later point, render this template with a parameter and *without* any additional network access:

```
$("#main").processTemplate(
  {'election': election_object}
);
```

## 3.2 Cryptography in the Browser with LiveConnect

JavaScript is a complete programming language in which it is possible to build a multi-precision integer library. Unfortunately, JavaScript performance for such computationally intensive operations is poor. Thankfully, it is possible in modern browsers to access the browser’s Java Virtual Machine from JavaScript using a technology called LiveConnect. This is particularly straightforward in Firefox, where one can write the following JavaScript code:

```
var a = new java.math.BigInteger(42);
document.write(a.toString());
```

and then, from JavaScript still, invoke all of Java’s `BigInteger` methods directly on the object. Modular exponentiation is a single call, `modPow()`, and El-Gamal encryption runs fast enough that it is close to imperceptible to the average user. LiveConnect is slightly more complicated to implement in Internet Explorer and Safari, though it can be done [18].

## 3.3 Additional Tricks

**Data URIs.** At times in the Helios protocol, we need to produce a printable receipt when the plaintext vote has not yet been cleared from memory. In order to open a new window ready for printing without network access, we use data URIs [14], URIs that contain information without requiring a network fetch:

```
<a target="_new"
   href="data:text/plain,Your%20Receipt...">
     receipt
</a>
```

**Dynamic Windows.** When data URIs are not available (e.g. Internet Explorer), we can open a new window using JavaScript, set its MIME type to `text/plain`, and dynamically write its content from the calling frame.

```
var receipt = window.open();
receipt.document.open("text/plain");
receipt.document.write(content);
receipt.document.close();
```

In Safari and Firefox, this approach yields a new window in a slightly broken state: the contents cannot be saved to disk. However, in Internet Explorer, the only browser that does not support Data URIs, the dynamic window creation works as expected. Thus, in Firefox and Safari, Helios uses Data URIs, and in Internet Explorer it uses dynamic windows.

**JSON.** As we expect that auditors will want to download election, voter, and bulletin board data for processing and verifying, we need a data format that is easy to parse in most programming languages, including JavaScript. XML is one possibility, but we found that JavaScript Object Notation (JSON) is easier to handle with far less parsing code. JSON allows for data representation using JavaScript lists and associative arrays. For example, a list of voters and their encrypted votes can be represented as:

```
[
  {'name' : 'Alice', 'vote' : '23423....'},
  {'name' : 'Bob', 'vote' : '823848....'},
  ...
]
```

Libraries exist in all major programming languages for parsing and generating this data format. In particular, the format maps directly to arrays and objects in JavaScript, lists and dictionaries in Python, lists and hashes in Ruby.

## 4 Helios System Description

We are now ready to discuss the details of the Helios system. We begin with a description of the back-end server architecture. We then consider the four use cases: creating an election, voting, tallying, and auditing.

### 4.1 Server Architecture

The Helios back-end is a Web application written in the Python programming language [17], running inside the CherryPy 3.0 application server, with a Lighttpd web server. All data is stored in a PostgreSQL database. All server-side logic is implemented in Python, with HTML templates rendered using the Cheetah Templating engine. Many back-end API calls return JSON data structures using the Simplejson library, and the voting booth server-side template is, in fact, a single-page web applications including JavaScript logic and jTemplate HTML/JavaScript templates.

**Application Software.** We use the Python Cryptography Toolkit for number theory utilities such as prime number and random number generation. We implemented our own version of El-Gamal in Python, given our specific need for re-encryption, which is typically not supported in cryptographic libraries. We note that improved performance could likely be gained from optimizing our first-pass implementation.

**Server Hardware.** We host an alpha version of the Helios software at `http://heliosvoting.org`. The server behind that URL is a virtual Ubuntu Linux server operated by SliceHost. For the tests performed in Section 5.3, we used a small virtual host with 256 megabytes of RAM and only a fraction of a Xeon processor, at a cost of $20/month. A larger virtual host would surely provide better performance, but we wish to show the practicality of Helios even with modest resources.

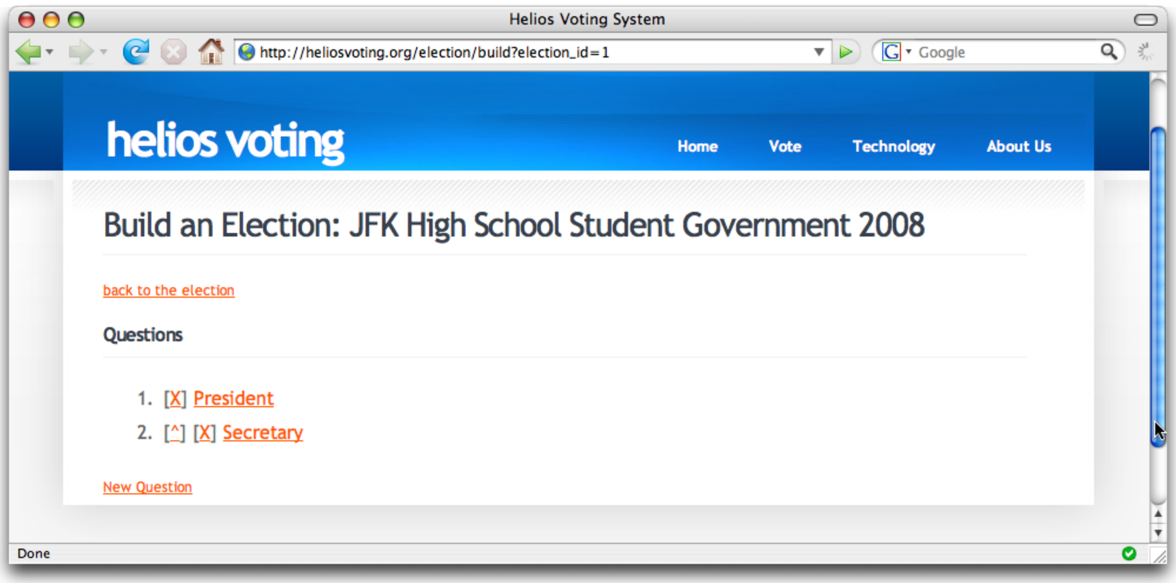
### 4.2 Creating an Election

Only registered Helios users can create elections. Registration is handled like most typical web sites:

- a user enters an email address, a name, and a desired password.
- an email with an embedded confirmation link is sent to the given email address.
- the user clicks on the confirmation link to activate his account.

A registered user then creates an election with an election name, a date and time when voting is expected to begin, and a date and time when voting is expected to end. Upon creation, Helios generates and stores a new El-Gamal keypair for the election. Only the public key is available to the registered user: Helios keeps the private key secret. The user who created the election is considered the *administrator*.

**Setting up the Ballot.** The election is then in "build mode," where the ballot can be prepared, reviewed, and tweaked by the administrative user, as shown in Figure 2. The user can log back in over multiple days to adjust any aspect of the ballot.



Build an Election: JFK High School Student Government 2008

back to the election

Questions

1. [X] President
2. [^] [X] Secretary

New Question

Figure 2: The Helios Election Builder lets an administrative user create and edit ballot questions in a simple web-based interface. The administrative user can log out and back in at any time to update the election.

**Managing Voters.** The administrative user can add, update, and remove voters at will, as shown in Figure 3. A voter is identified by a name and an email address, and is specific to a given election. Helios generates a random 10-character password automatically for each voter. At any time, the administrator can email voters using the Helios administrative interface. These emails will automatically contain the voter's password, though the administrator will not see this password at any time.

**Freezing the Election.** When ready, the administrative user *freezes the election*, at which point the voter list, the election start and end dates, and the ballot details become immutable and available for download in JSON form. The administrative user receives an email from Helios with the SHA1 hash of this JSON object. The election is ready for voters to cast ballots. The administrative user will typically email voters using the Helios administrative interface to let them know that the polls are open.

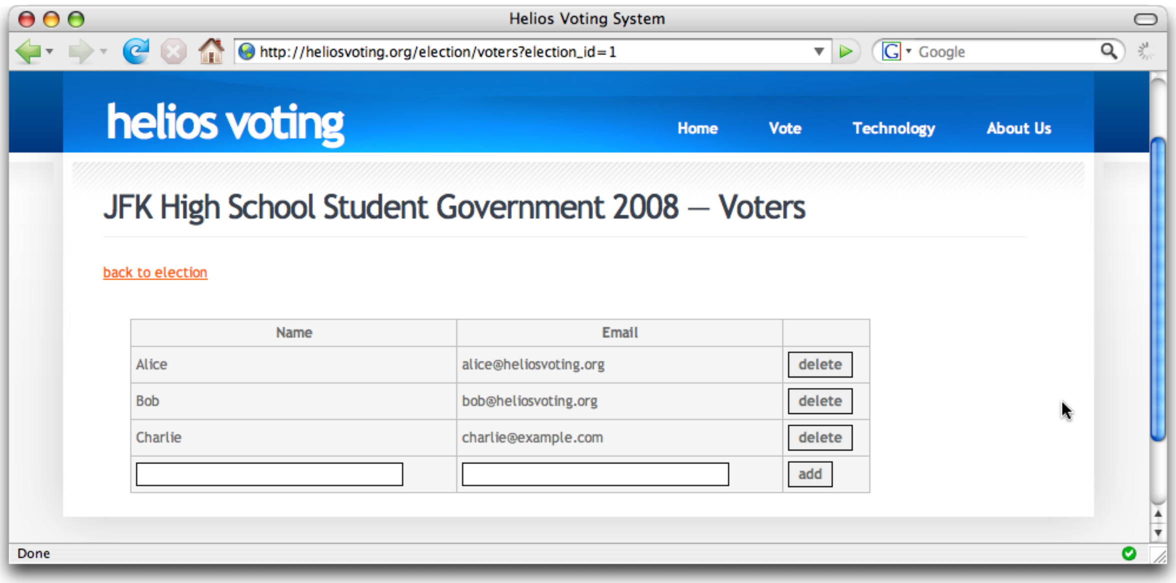
## 4.3 Voting

Alice, a voter in a Helios election, receives an email letting her know that the polls are open. This email contains her username (i.e. her email address), her election-specific password, the SHA1 hash of the election parameters, and the URL that directs her to the Helios voting booth, as illustrated in Figure 4. It is important to note that this URL does not contain any identifying information: it only identifies the election, as per the vote-casting protocol in Section 2.1.

**The Voting Booth.** When Alice follows the voting booth URL, Helios responds with a single-page web application. This application, now running in Alice's browser, displays a "loading..." message while it downloads the election parameters and templates, including the El-Gamal public key and questions. The page then displays the election hash prominently, and indicates that no further network connections will be made until Alice submits her encrypted ballot. (Alice can set her browser to "offline" mode to enforce this.) Every transition is then handled by a local JavaScript function call and its associated templates. Importantly, the JavaScript code can decide precisely what state to maintain and what state to discard: the "back" button is not relevant. This is illustrated in Figure 5.

**Filling in the Ballot.** Alice can then fill in the ballot, selecting the checkbox by each desired candidate name, using the "next" and "previous" buttons to navigate between questions. Each click is handled by JavaScript code which records Alice's choices in the local JavaScript scope. If Alice tries to close her browser or navigate to a different URL, she receives a warning that her ballot will be cleared.

**Sealing.** After Alice has reviewed her options, she can choose to "seal" her ballot, which triggers the JavaScript code to encrypt her selection with computationally intensive operations performed via LiveConnect. The SHA1 hash of the resulting ciphertext is then displayed, as shown in Figure 6.

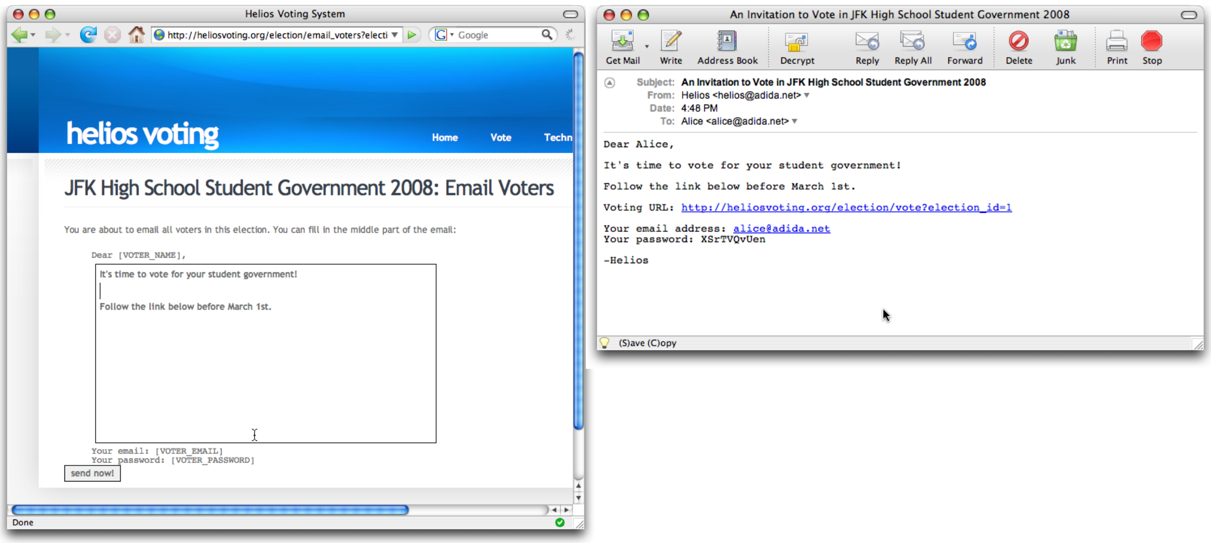


JFK High School Student Government 2008 — Voters

back to election

| Name | Email | |
|---|---|---|
| Alice | alice@heliosvoting.org | delete |
| Bob | bob@heliosvoting.org | delete |
| Charlie | charlie@example.com | delete |
| | | add |

Figure 3: The Helios voter management interface.



JFK High School Student Government 2008: Email Voters

You are about to email all voters in this election. You can fill in the middle part of the email:

```
Dear [VOTER_NAME],
```

It's time to vote for your student government!

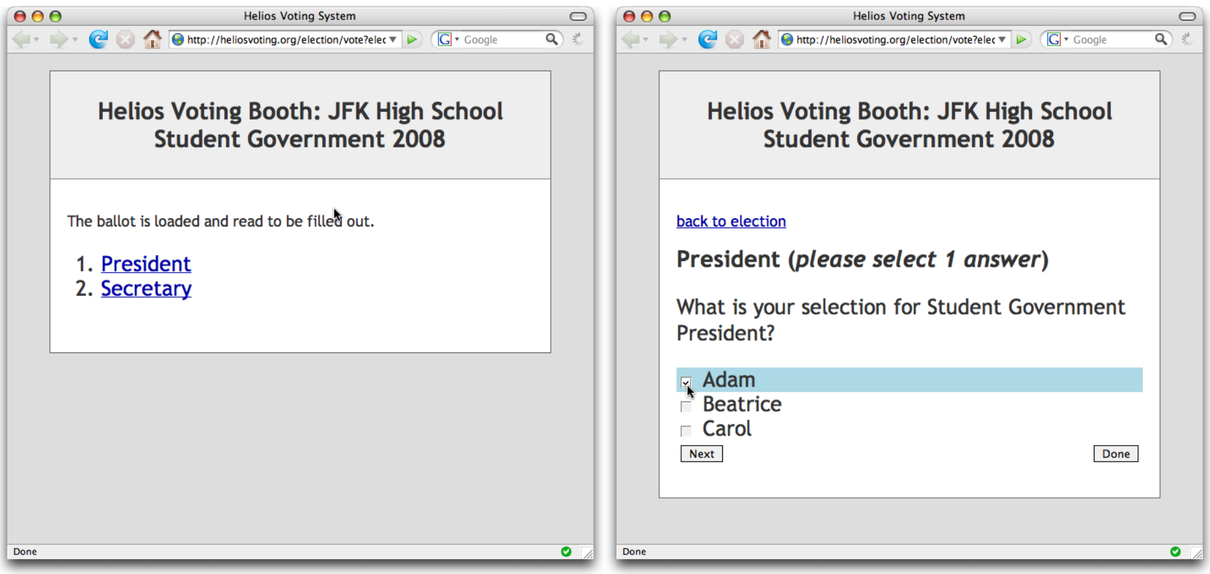
Follow the link below before March 1st.

```
Your email: [VOTER_EMAIL]
Your password: [VOTER_PASSWORD]
```

Subject: An Invitation to Vote in JFK High School Student Government 2008
From: Helios <helios@adida.net>
Date: 4:48 PM
To: Alice <alice@adida.net>

```
Dear Alice,

It's time to vote for your student government!

Follow the link below before March 1st.

Voting URL: http://heliosvoting.org/election/vote?election_id=1

Your email address: alice@adida.net
Your password: XSrTVQvUen

-Helios
```

Figure 4: The administrative user can send emails to all voters. Each voter receives her password, which the administrative user does not see.


Helios Voting System

http://heliosvoting.org/election/vote?elec

Helios Voting Booth: JFK High School Student Government 2008

The ballot is loaded and read to be filled out.

1. President
2. Secretary

Done

Helios Voting System

http://heliosvoting.org/election/vote?elec

Helios Voting Booth: JFK High School Student Government 2008

back to election

**President (*please select 1 answer*)**

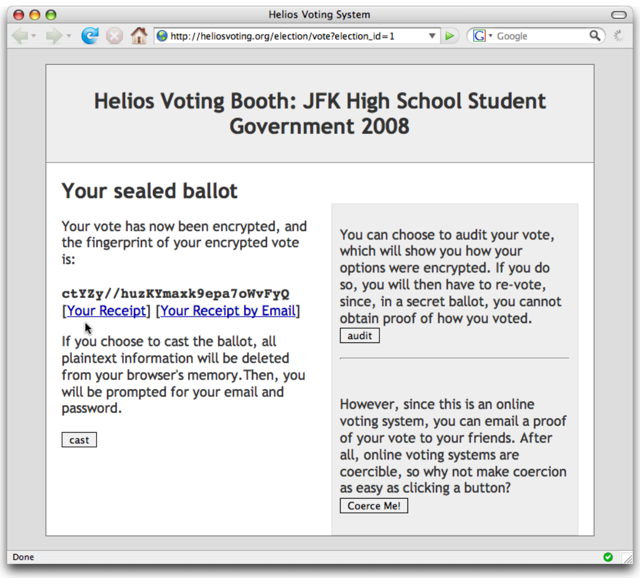
What is your selection for Student Government President?

[✓ Adam]
Beatrice
Carol

Next Done

Done

Figure 5: The Helios Voting Booth.


Helios Voting System

http://heliosvoting.org/election/vote?election_id=1

Helios Voting Booth: JFK High School Student Government 2008

**Your sealed ballot**

Your vote has now been encrypted, and the fingerprint of your encrypted vote is:

**ctYZy//huzKYmaxk9epa7oWvFyQ**
[Your Receipt] [Your Receipt by Email]

If you choose to cast the ballot, all plaintext information will be deleted from your browser's memory.Then, you will be prompted for your email and password.

cast

You can choose to audit your vote, which will show you how your options were encrypted. If you do so, you will then have to re-vote, since, in a secret ballot, you cannot obtain proof of how you voted.

audit

However, since this is an online voting system, you can email a proof of your vote to your friends. After all, online voting systems are coercible, so why not make coercion as easy as clicking a button?

Coerce Me!

Done

Figure 6: Sealing a Helios ballot.

**Auditing.** Alice can opt to audit her ballot with the "Audit" button, in which case the JavaScript code reveals the randomness used in encrypting Alice's choices. Alice can save this data to disk and run her own code to ensure the encryption was correct, or she can use the Python Ballot Encryption Verification (BEV) program provided by Helios.

Once Alice chooses to audit her ballot and the auditing information is rendered, the JavaScript code clears its encrypted ballot data structures and returns Alice to the confirmation screen, where she can either update her choices or choose to seal her options again with different randomness and thus a different ciphertext.

**Casting.** If Alice chooses instead to cast her ballot, the JavaScript code clears the plaintext and randomness from its scope, and presents Alice with a login prompt for her email address and password. (If Alice had set her browser to "offline" mode, she should bring it back online now that all plaintext information is cleared.) When Alice submits her login information, the JavaScript code intercepts the form submission and submits the email, password, and encrypted vote in a background call, so that any errors, e.g. a mistyped password, can be reported without clearing the JavaScript scope and thus the encrypted ballot. When a success code is returned by the Helios server, the JavaScript code can clear its entire scope and display a success message. On the server side, Helios emails Alice with a confirmation of her encrypted vote, including its SHA1 hash.

**Coerce Me!** As explained in Section 2, Helios provides a "Coerce Me!" button to make it clear that online voting is inherently coercible. This button appears after ballot sealing, next to the "audit" and "cast" options. When clicked, Helios opens up a new window with a `mailto:` URL that triggers Alice's email client to open a composition window containing the entire ballot information, including plaintext and randomness that prove how the ciphertext was formed. Unlike the "Audit" step, which forces Alice to create a new ciphertext, "Coerce Me!" allows Alice to continue and cast that very same encrypted vote for which she obtained proof of encryption. The distinction between these two steps highlights the difference between a coercion-free auditing process that could potentially be used with in-person voting, and the inherent coercibility of online-only voting which is made more explicit with the "Coerce Me!" button.

## 4.4 Anonymization

Once the voting period ends, Helios enables the anonymization, decryption, and proof features for the administrative user. Selecting "shuffle" will begin the re-encryption and permutation process. Then, selecting "shuffle proof" will trigger the mixnet proof with 80 shadow mixes. The administrative user can then opt for "decrypt", which will decrypt the shuffled ciphertexts, and "decrypt proof", which will generate proofs for each such decryption. Finally, the administrative user can select "tally" to count up the decrypted votes.

All of these operations are performed on the server side, in Python code. The results are stored in the database and made available for download in JSON form. Once all proofs are generated and the result is tallied, the server deletes the permutation, randomness, and secret key for that election. All that is left is the encrypted votes, their shuffling, the resulting decryptions, and the publicly verifiable proofs of integrity. The entire election can still be verified, though no further proofs can be generated.

## 4.5 Auditing

Helios provides two verification programs, one for verifying a single encrypted vote produced by the ballot preparation system with the "audit" option selected, and another for verifying the shuffling, decryption, and tallying of an entire election. Both programs are written in Python using the Simplejson library for JSON processing, but otherwise only raw Python operations.

**Verifying a Single Vote.** The Ballot Encryption Verification program takes as input the JSON data structure returned by the voting booth audit process. This data structure contains a plaintext ballot, its ciphertext, the randomness used to encrypt it, and the election ID. The program downloads the election parameters based on the election ID and outputs:

- the hash of the election, which the voter can check against that displayed by the voting booth,
- the hash of the ciphertext, which the voter can check against the receipt she obtained before requesting an audit,
- the verified plaintext of the ballot.

**Verifying an Election.** The Election Tallying Verification program takes, as input, an election ID. It downloads the election parameters, the bulletin board of cast votes, shuffled votes, shuffle proofs, decrypted votes, and decryption proofs. The verification program checks all proofs, then re-performs the tally based on the decryptions. It eventually outputs the list of voters and their respective encrypted ballot hashes, plus the verified tally. This information can be reposted by the auditor, so that if enough auditors check and re-publish the cast ballot hashes and tally, participants can be confident that their vote was correctly captured, and that the tally was correctly performed.

## 5 Discussion

Helios is simpler than most cryptographic voting protocols because it focuses on proving integrity. As a compromise, Helios makes weaker guarantees of privacy. In this section, we review in greater detail the type of election for which we expect this compromise to be appropriate, as well as the security model, performance metrics, and future extensions we can make to improve Helios on both fronts.

### 5.1 The Need for Verifying Elections with Low Coercion Risk

It is legitimate to question whether there truly exist elections that require the high levels of verifiability afforded by cryptography, while eschewing coercion-resistance altogether. In fact, we believe that, for a number of online communities that rarely or never meet in the same physical place:

1. coercion-resistance is futile from the start, given the remote nature of the voting process, and
2. cryptographic end-to-end verifiability is the *only* viable means of ensuring *any* level of integrity.

Specifically, with respect to the auditing argument, how could a community member remotely verify anything at all pertaining to the integrity of an election process? Open-source software is insufficient: the voter doesn't know *which* software is actually running on the election server, short of deploying hardware-rooted attestation. Physical observation of a chain-of-custody process is already ruled out by the online-only nature of the community. Cryptographic verifiability, though it seems stronger than absolutely necessary, is the only viable option when only the public inputs and outputs—never the "guts"—of the voting process can be truly observed. Cryptographic auditing may be a big hammer, but it is the only hammer.

For the same reason, we believe the pedagogical value of a system like Helios is particularly strong. The contrast between classic and open-audit elections is particularly apparent in this online setting. With Helios, the voter's ability is transformed, from entirely powerless and forced to trust a central system, to empowered with the ability to ensure that one's vote was correctly captured and tallied, without trusting anyone.

### 5.2 Security Model & Threats

We accept the risk that, if someone compromises the Helios server before the end of an election, the secrecy of individual ballots may be compromised. On the other hand, we claim that, assuming enough auditors, even a fully corrupted Helios cannot cheat the election result without a high chance of getting caught. We now explore various attacks and how we expect them to be handled.

**Incorrect Shuffling or Decryption.** A corrupt Helios server may attempt to shuffle votes incorrectly or decrypt shuffled votes incorrectly. Given the overwhelming probability of catching these types of attacks via cryptographic verification, it takes only one auditor to detect this kind of tampering.

**Changing a Ballot or Impersonating a Voter.** A corrupt Helios may substitute a new ciphertext for a voter, replacing his cast vote or injecting a vote when a voter doesn't cast one in the first place. Even if the ballot submission server is eventually hosted separately and distributed among trustees, a corrupt Helios server knows the username and password for all users, and can thus easily authenticate and cast a ballot on behalf of a user. In this case, all of the shuffling and decryption verifications will succeed, because the corruption occurs before the encryption step.

In the current implementation of Helios, we hope to counter these attacks through extensive auditing. Previous analyses [7] have shown that it takes only a small random sample of voters who verify their vote to defeat this kind of attack. To encourage voters to audit their votes, we created the Election Tallying Verification program, available in well commented source form. The Election Tallying Verification program outputs a copy of all cast ballots, so that auditors can post this information independently. We expect multiple auditors to follow this route and re-publish the complete list of encrypted ballots along with their re-computed election outcome. This auditing may include re-contacting individual voters and asking them to verify the hash of their cast encrypted ballot. We also expect that a large majority of voters, maybe all voters, in fact, will answer at least one auditor who prompts them to verify their cast encrypted vote.

**Corrupting the Ballot.** A corrupt Helios may present a corrupt ballot to Alice, making her believe that she's selecting one candidate when actually she is voting for another. This kind of attack would defeat the hashed-vote bulletin-board verification, even with multiple auditors, since Alice receives an entirely incorrect receipt during the ballot casting process. Helios mitigates this risk by authenticating users only *after* the ballot has been filled out, so users cannot be individually targeted with corrupt ballots as easily. However, a corrupt Helios may authenticate voters first (voters may not notice), or use other information (e.g. IP address) to identify voters and target certain victims for ballot corruption.

To counter this attack, we provide the Ballot Encryption Verification program, again in source form for auditors to verify. This program can be run by individual voters when they choose to audit a handful of votes before they choose to truly cast one. Alternatively, auditors, even auditors who are not eligible to vote in the election, can prepare ballots and audit them at will.

**Auditing is Crucial.** It should be clear from these descriptions that Helios counters attacks through the power of auditing. In addition to the raw tally, Helios publishes a list of voter names and corresponding encrypted votes. Helios then provides supporting evidence for the tally, given the cast encrypted votes, in the form of a mixnet-and-decryption proof. Verification programs are available in source form for anyone to review the integrity of the results.

However, only the individual voters can check the validity of the cast encrypted ballots. It is expected that multiple auditors will check the proof and, when satisfied, republish the tally *and* the list of cast encrypted ballots, where voters can check that their ballot was correctly recorded. Helios ensures that, if a large majority of voters verifies their vote, then the outcome is correct. However, if voters do not verify their cast ballot, Helios does not provide any verification beyond classic voting systems.

These expectations are somewhat tautological: voter-verified elections function only when at least some fraction of the voters are willing to participate in the verification process made available to them. Elections can be made *verifiable*, but only voters can actually verify that their *secret* ballot was correctly recorded.

## 5.3 Performance

For all performance measurements, we used the server hardware described in the previous section, and, on the client side, a 2.2Ghz Macintosh laptop running Firefox 2 over a home broadband connection. We note that performance of Firefox 2 was greatly increased when running on virtualized Linux on the same laptop, indicating that our measurements are likely a worst-case scenario given platform-specific performance peculiarities of Firefox.

**Java Virtual Machine Startup.** The Java Virtual Machine requires startup time. Our rough measurements indicate anywhere between 500ms and 1.5s on our client machine. During this time, the browser appears to freeze and user input is suspended. To an uninformed user, this is a usability impediment which will require further user testing. That said, it is a behavior we can easily warn users about before starting up the Ballot Preparation System, and because this happens only once per user session – not once per ballot – it is not too onerous.

**Timing Measurements.** We experimented with a 2-question election and 500 voters. All timings were performed a sufficient number of times to obtain a stable average mostly free of testing noise. Note that time measurements that pertain to a set of ballots are expected to scale linearly with the number ballots and the number of questions in the election. Our results are presented in Figure 7.

| **Operation** | **Time** |
|---|---|
| Ballot Encryption, in browser $\|p\| = 1024$ bits | 300ms |
| Shuffling, on server | 133 s |
| Shuffle Proof, on server | 3 hours |
| Decryption, on server | 71 s |
| Decryption Proof, on server | 210 s |
| Complete Audit, on client | 4 hours |

Figure 7: Timing Measurements

**The Big Picture.** It takes only a few minutes of computation to obtain results for a 500-voter election. The shuffle proof and verification steps require a few hours, and are thus, by far, the most computation-intensive portions of the process. We note that both of these steps are highly parallelizable and thus could be significantly accelerated with additional hardware.

## 5.4 Extensions

There are many future directions for Helios.

**Support for Other Types of Election.** Helios currently supports only simple elections where Alice selects 1 or more out of the proposed candidates. Adding write-ins and rank-based voting, as well as the associated tallying mechanisms, could prove useful. Helios may also eventually offer homomorphic-based tabulation, as they are often easier to explain and verify, though they would made greater demands of browser-based cryptography.

**Browser-Based Verification.** The current verification process for the ballot encryption step is a bit tedious, requiring the use of a browser and a Python program. We could write a JavaScript-only verification program that could be provided directly by auditors while running entirely in the voter's browser to check that Helios is delivering authentic ballots. There are some issues to deal with, notably cross-domain requests, but it does seem possible and desirable to accomplish browser-only ballot encryption verification.

Similarly, it is certainly possible to audit an entire election using JavaScript and LiveConnect for computationally intensive operations. Letting auditors deliver the source code for these verification programs would allow any voter to audit the entire process straight from their browser.

**Distributing the Shuffling and Decryption.** For improved privacy guarantees, Helios can be extended to support shuffling and decryption by multiple trustees. The Helios server would then only focus on providing the bulletin board and voting booth functionality. Trustees would be provided with standalone Python programs that perform threshold key generation, partial shuffling and threshold decryption. They could individually audit the program's source code. With these extensions, Helios would resemble classic cryptographic voting protocols more closely, and would provide stronger privacy guarantees.

**Improving Authentication.** Currently, our protocol requires that most voters audit their cast ballot, otherwise the Helios server could impersonate voters and change the election outcome. Future version of Helios should consider offloading authentication to a separate authentication service. If feasible with browser-based cryptography, Helios should use digital signatures to authenticate each ballot in a publicly verifiable manner.

## 6 Related Work

There is a plethora of theoretical cryptographic voting work reviewed and cited in [11, 4]. We do not attempt to re-document this significant body of work here.

**Open-audit voting implementations.** There are only a small handful of notable open-audit voting implementations. VoteHere's advanced protocols for mixnets and coercion-free ballot casting [3] have been implemented and deployed in test environments. The Punchscan voting system [2] has also been implemented and used in a handful of real student government elections, with video evidence available for all to see.

**Browser-based cryptography.** Cryptographic constructs have been implemented in browser-side code in many different settings. In the late 1990s, Hushmail began providing web-based encrypted email using a Java applet. A couple of years later, George Danezis showed how to use LiveConnect for fast JavaScript-based cryptography, and the EVOX voting project [12] used similar technology to encrypt votes in a blind-signature-based scheme. The Stanford SRP project [18] also uses LiveConnect for browser-based cryptography and indicates how one can get LiveConnect to work in browsers other than Firefox. The recent Clipperz Crypto Library [1] provides web-based cryptography in pure JavaScript, including a multi-precision integer library.

## 7 Conclusion

Helios is the first publicly available implementation of a web-based open-audit voting system. It fills an interesting niche: elections for small clubs, online communities, and student governments need trustworthy elections without the significant overhead of coercion-freeness. We hope that Helios can be a useful educational resource for open-audit voting by providing a valuable service – outsourced, verifiable online elections – that could not be achieved without the paradigm-shifting contributions of cryptographic verifiability.

## 8 Acknowledgements

The author would like to thank Ronald L. Rivest, Shai Halevi, Chris Peikert, Susan Hohenberger, Alon Rosen, Steve Weis, and Greg Morrisett for alpha testing the Helios system, Adam Barth for pointing out the dynamic window creation technique for the Internet Explorer work-around, and the Usenix Security reviewers for insightful suggestions in improving and crystalizing the presentation of this work.

## References

[1] Clipperz. `http://clipperz.org`, last viewed on January 30th, 2008.

[2] PunchScan. `http://punchscan.org`, last viewed on January 30th, 2008.

[3] VoteHere. `http://votehere.com`, last viewed on January 30th, 2008.

[4] Ben Adida. *Advances in Cryptographic Voting Systems*. PhD thesis, August 2006. `http://ben.adida.net/research/phd-thesis.pdf`.

[5] Josh Benaloh. Simple Verifiable Elections. In *EVT '06, Proceedings of the First Usenix/ACCURATE*

*Electronic Voting Technology Workshop, August 1st 2006, Vancouver, BC,Canada.*, 2006. Available online at `http://www.usenix.org/events/evt06/tech/`.

[6] Josh Benaloh. Ballot Casting Assurance via Voter-Initiated Poll Station Auditing. In *EVT '07, Proceedings of the Second Usenix/ACCURATE Electronic Voting Technology Workshop, August 6th 2007, Boston, MA, USA.*, 2007. Available online at `http://www.usenix.org/events/evt07/tech/`.

[7] C. Andrew Neff. Election Confidence. `http://www.votehere.com/papers/ElectionConfidence.pdf`, last viewed on January 30th, 2008.

[8] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.

[9] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[10] Google. Gmail. `http://gmail.com`.

[11] Dimitris Gritzalis, editor. *Secure Electronic Voting*. Kluwer Academic Publishers, 2002.

[12] Mark Herschberg. Secure electronic voting over the world wide web. Master's thesis, May 1997. `http://groups.csail.mit.edu/cis/voting/herschberg-thesis/`.

[13] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine, editors, *WPES*, pages 61–70. ACM, 2005.

[14] L. Masinter. The Data URL Scheme. `http://tools.ietf.org/html/rfc2397`, last viewed on January 30th, 2008.

[15] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security. November 6-8, 2001, Philadelphia, Pennsylvania, USA.*, pages 116–125. ACM, 2001.

[16] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme - a practical solution to the implementation of a voting booth. In *EUROCRYPT*, pages 393–403, 1995.

[17] Guido van Rossum. The Python Programming Language. `http://python.org`, last viewed on January 30th, 2008.

[18] Thomas D. Wu. The secure remote password protocol. In *NDSS*. The Internet Society, 1998. `http://srp.stanford.edu/`, last visited on January 30th, 2008.

# VoteBox: a tamper-evident, verifiable electronic voting system

Daniel Sandler Kyle Derr Dan S. Wallach
*Rice University*
{dsandler,derrley,dwallach}@cs.rice.edu

## Abstract

*Commercial electronic voting systems have experienced many high-profile software, hardware, and usability failures in real elections. While it is tempting to abandon electronic voting altogether, we show how a careful application of distributed systems and cryptographic techniques can yield voting systems that surpass current systems and their analog forebears in trustworthiness* and *usability. We have developed the VoteBox, a complete electronic voting system that combines several recent e-voting research results into a coherent whole that can provide strong end-to-end security guarantees to voters. VoteBox machines are locally networked and all critical election events are broadcast and recorded by every machine on the network. VoteBox network data, including encrypted votes, can be safely relayed to the outside world in real time, allowing independent observers with personal computers to validate the system as it is running. We also allow any voter to challenge a VoteBox, while the election is ongoing, to produce proof that ballots are cast as intended. The VoteBox design offers a number of pragmatic benefits that can help reduce the frequency and impact of poll worker or voter errors.*

## 1 Introduction

Electronic voting is at a crossroads. Having been aggressively deployed across the United States as a response to flawed paper and punch-card voting in the 2000 U.S. national election, digital-recording electronic (DRE) voting systems are themselves now seen as flawed and unreliable. They have been observed in practice to produce anomalies that may never be adequately explained—undervotes, ambiguous audit logs, choices "flipping" before the voter's eyes. Recent independent security reviews commissioned by the states of California and Ohio have revealed that every DRE voting system in widespread use has severe deficiencies in design and implementation, exposing them to a wide variety of vulnerabilities; these systems were never engineered to be secure. As a result, many states are now decertifying or restricting the use of DRE systems.

Consequently, DREs are steadily being replaced with systems employing optical-scan paper ballots. Op-scan systems still have a variety of problems, ranging from accessibility issues to security flaws in the tabulation systems, but at least the paper ballots remain as evidence of the voter's original intent. This allows voters some confidence that their votes can be counted (or at least recounted) properly. However, as with DRE systems, if errors or tampering occur anywhere in this process, there is no way for voters to independently verify that their ballots were properly tabulated.

Regardless, voters subjectively prefer DRE voting systems [15]. DREs give continuous feedback, support many assistive devices, permit arbitrary ballot designs, and so on. Furthermore, unlike vote-by-mail or Internet voting, DREs, used in traditional voting precincts, provide privacy, protecting voters from bribery or coercion. We would ideally like to offer voters a DRE-style voting system with additional security properties, including:

1. Minimized software stack
2. Resistance to data loss in case of failure or tampering
3. Tamper-evidence: a record of election day events that can be believably audited
4. End-to-end verifiability: votes are cast as intended and counted as cast

The subject of this paper is the VoteBox, a complete electronic voting system that offers these essential properties as well as a number of other advantages over existing designs. Its user interface is built from pre-rendered graphics, reducing runtime code size as well as allowing the voter's exact voting experience to be examined well before the election. VoteBoxes are networked in a precinct and their secure logs are intertwined and replicated, providing robustness and auditability in case of failure, misconfiguration, or tampering. While all of these techniques have been introduced before, the novelty of this work lies in our integration of these parts to achieve our architectural security goals.

Notably, we use a technique adapted from Benaloh's work on voter-initiated auditing [4] to gain end-to-end verifiability. Our scheme, which we term *immediate ballot challenge*, allows auditors to compel any active voting machine to produce proof that it has correctly captured the voter's intent. With immediate challenges, every single ballot may potentially serve as an election-day test of a VoteBox's correctness. We believe that the VoteBox architecture is robust to the kinds of failures that commonly occur in elections and is sufficiently auditable to be trusted with the vote.

In the next section we will present background on the electronic voting problem and the techniques brought to bear on it in our work. We expand on our design goals and describe our VoteBox architecture in Section 3, and share details of our implementation in Section 4. The paper concludes with Section 5.

## 2 Background

### 2.1 Difficulties with electronic voting

While there have been numerous reports of irregularities with DRE voting systems in the years since their introduction, the most prominent and indisputable problem concerned the ES&S iVotronic DRE systems used by Sarasota County, Florida, in the November 2006 general election. In the race for an open seat in the U.S. Congress, the margin of victory was only 369 votes, yet over 18,000 votes were officially recorded as "undervotes" (i.e., cast with no selection in this particular race). In other words, 14.9% of the votes cast on Sarasota's DREs for Congress were recorded as being blank, which contrasts with undervote rates of 1–4% in other important national and statewide races. While a variety of analyses were conducted of the machines and their source code [18, 19, 51], the official loser of the election continued to challenge the results until a Congressional investigation failed to identify the source of the problem [3]. Whether the ultimate cause was mechanical failure of the voting systems or poor human factors of the ballot design, there is no question that these machines failed to accurately capture the will of Sarasota's voters [2, 14, 20, 25, 34, 36, 37, 50].

While both security flaws and software bugs have received significant attention, a related issue has also appeared numerous times in real elections using DREs: operational errors and mistakes. In a 2006 primary election in Webb County, Texas—the county's first use of ES&S iVotronic DRE systems—a number of anomalies were discovered when, as in Sarasota, a close election led to legal challenges to the outcome [46]. Test votes were accidentally counted in the final vote tallies, and some machines were found to have been "cleared" on election day, possibly erasing votes. More recently, in the January, 2008 Republican presidential primary in South Carolina, several ES&S iVotronic systems were incorrectly configured subsequent to pre-election testing, resulting in those machines being inoperable during the actual election. "Emergency" paper ballots ran out in many precincts and some voters were told to come back later [11].

All of these real-world experiences, in conjunction with recent highly critical academic studies, have prompted a strong backlash against DRE voting systems or even against the use of computers in any capacity in an election. However, computers are clearly beneficial.

Clearly, computers cannot be trusted to be free of tampering or bugs, nor can poll workers and election officials be guaranteed to always operate special-purpose computerized voting systems as they were intended to be used. Our challenge, then, is to reap the benefits that computers can offer to the voting process without being a prisoner to their costs.

### 2.2 Toward software independence

Recently, the notion of *software independence* has been put forth by Rivest and other researchers seeking a way out of this morass:

> *A voting system is* software-independent *if an undetected change or error in its software cannot cause an undetectable change or error in an election outcome.* [41]

Such a system produces results that are verifiably correct or incorrect irrespective of the system's implementation details; any software error, whether malicious or benign, cannot yield an erroneous output masquerading as a legitimate cast ballot.

Conventionally, the only way to achieve true software independence is to allow the voter to directly inspect, and therefore confirm to be correct, the actual cast vote record. Since we cannot give voters the ability to read bits off a flash memory card, nor can we expect them to mentally perform cryptographic computations, we are limited in practice to paper-based vote records, which can be directly inspected.

Optical-scan voting systems, in which the voter marks a piece of paper that is both read immediately by an electronic reader/tabulator and reserved in case of a manual audit, achieve this goal at the cost of sacrificing some of the accessibility and feedback afforded by DREs. The voter-verifiable paper audit trail (VVPAT) allows a DRE to *create* a paper record for the voter's inspection and for use in an audit, but it has its own problems. Adding printers to every voting station dramatically increases the mechanical complexity, maintenance burden, and failure rate of those machines. A report on election problems in the 2006 primary in Cuyahoga County, Ohio found that 9.6% of VVPAT records were destroyed, blank, or "compromised in some way" [23, p. 93].

Even if the voter's intent survives the printing process, the rolls of thermal paper used by many current VVPAT printers are difficult to audit by hand quickly and accurately [22]. It is also unclear whether voters, having already interacted with the DRE and confirmed their choices there, will diligently validate an additional paper record. (In the same Cuyahoga primary election, a different report found that voters in fact did not know they were supposed to open a panel and examine the printed tape underneath [1, p. 50].)

### 2.2.1 Reducing the trusted computing base

While the goal of complete software independence is daunting, the state of the art in voting research approaches it by drawing a line around the set of functions that are essential to the correctness of the vote and aggressively evicting implementation from that set. If assurance can come from reviewing and auditing voting software, then it should be easier to review and ultimately gain confidence in a smaller software stack.

Pre-rendered user interface (PRUI) is an approach to reducing the amount of voting software that must be reviewed and trusted [53]. Exemplified by Pvote [52], a PRUI system consists of a ballot definition and a software system to present that ballot. The ballot definition comprises a state machine and a set of static bitmap images corresponding to those states; it represents what the voter will see and interact with. The software used in the voting machine acts as a virtual machine for this ballot "program." It transitions between states and sends bitmaps to the display device based on the voter's input (e.g., touchscreen or keypad). The voting VM is no longer responsible for text rendering or layout of user interface elements; these tasks are accomplished long in advance of election day when the ballot is defined by election officials.

A ballot definition of this sort can be audited for correctness independently of the voting machine software *or* the ballot preparation software. Even auditors without knowledge of a programming language can follow the state transitions and proofread the ballot text (already rendered into pixels). The voting machine VM should still be examined by software experts, but this code—critical to capturing the user's intent—is reduced in size and therefore easier to audit. Pvote comprises just 460 lines of Python code, which (even including the Python interpreter and graphics libraries) compares favorably against current DREs: the AccuVote TS involves over 31,000 lines of C++ running atop Windows CE [52]. The system we describe in Section 3 applies the PRUI technique to reduce its own code footprint.

Sastry et al. [47] describe a system in which program modules that must be trusted are forced to be small and clearly compartmentalized by dedicating a separate computer to each. The modules operate on isolated CPUs and memory, and are connected with wires that may be observed directly; each module may therefore be analyzed and audited independently without concern that they may collude using side channels. Additionally, the modules may be powered off and on between voters to eliminate the possibility of state leaking from voter to voter. (Section 4.1 shows how we approximate this idea in software.)

### 2.2.2 The importance of audit logs

Even trustworthy software can be misused, and this problem occurs with unfortunate regularity in the context of electronic voting. We expect administrators to correctly deploy, operate, and maintain large installations of unfamiliar computer systems. DRE vendors offer training and assistance, but on election day there is typically very little time to wait for technical support while voters queue up.

In fact, the operational and procedural errors that can (and do) occur during elections is quite large. Machines unexpectedly lose power, paper records are misplaced, hardware clocks are set wrong, and test votes (see §2.2.3 below) are mingled with real ballots. Sufficient trauma to a DRE may result in the loss of its stored votes.

In the event of an audit or recount, comprehensive records of the events of election day are essential to establishing (or eroding) confidence in the results despite these kinds of election-day mishaps. Many DREs keep electronic audit logs, tracking election day events such as "the polls were opened" and "a ballot was cast," that would ideally provide this sort of evidence to *post facto* auditing efforts. Unfortunately, current DREs entrust each machine with its own audit logs, making them no safer from failure or accidental erasure than the votes themselves. Similarly, the audit logs kept by current DREs offer no integrity safeguards and are entirely vulnerable to attack; any malicious party with access to the voting machine can trivially alter the log data to cover up any misdeeds.

The AUDITORIUM [46] system confronts this problem by using techniques from distributed systems and secure logging to make audit logs into believable records. All voting machines in a polling place are connected in a private broadcast network; every election event that would conventionally be written to a private log is also "announced" to every voting machine on the network, each of which *also* logs the event. Each event is bound to its originator by a digital signature, and to earlier events from other machines via a *hash chain*. The aggressive replication protects against data loss and localized tampering; when combined with hash chains, the result is a hash mesh [48] encompassing every event in the polling place. An attacker (or an accident) must now successfully compromise every voting machine in the polling place in order to escape detection. (In Section 3 we describe how VoteBox uses and extends the Auditorium voting protocol.)

#### 2.2.3 Logic and accuracy testing; parallel testing

Regrettably, the conventional means by which voting machines are deemed trustworthy is through testing. Long before election day, the certification process typically involves some amount of source code analysis and testing by "independent testing authorities," but these processes have been demonstrably ineffective and insufficient. *Logic and accuracy* (L&A) testing is a common black-box testing technique practiced by elections officials, typically in advance of each election. L&A testing typically takes the form of a mock election: a number of votes are cast for different candidates, and the results are tabulated and compared against expected values. The goal is to increase confidence in the predictable, correct functioning of the voting systems on election day.

Complementary to L&A is *parallel* testing, performed on election day with a small subset of voting machines selected at random from the pool of "live" voting systems. The units under test are sequestered from the others; as with L&A testing, realistic votes are cast and tallied. By performing these tests on election day with machines that would otherwise have gone into service, parallel testing is assumed to provide a more accurate picture of the behavior of other voting machines at the same time.

The fundamental problem with these tests is that they are artificial: the conditions under which the test is performed are not identical to those of a real voter in a real election. It is reasonable to assume that a malicious piece of voting software may look for clues indicating a testing situation (wrong day; too few voters; evenly-spread voter choices) and behave correctly only in such cases. A software bug may of course have similar behavior, since faulty DREs may behave arbitrarily. We must also take care that a malicious poll worker cannot signal the testing condition to the voting machine using a covert channel such as a "secret knock" of user interface choices.

Given this capacity to "lay low" under test, the problem of fooling a voting machine into believing it is operating in a live vote-capture environment is paramount [26]. Because L&A testing commonly makes explicit use of a special code path, parallel testing is the most promising scenario. It presents its own unique hazard: if the test successfully simulates an election-day environment, any votes captured under test will be indistinguishable from legitimate ballots cast by real voters, so special care must be taken to keep these votes from being included in the final election tally.

### 2.3 Cryptography and e-voting

Many current DREs attempt to use encryption to protect the secrecy and integrity of critical election data; they universally fail to do so [6, 8, 24, 32]. Security researchers have proposed two broad classes of cryptographic techniques that go beyond simple encryption of votes (symmetric or public-key) to provide end-to-end guarantees to the voter. One line of research has focused on encrypting whole ballots and then running them through a series of mix-nets [9] that will re-encrypt and randomize ballots before they are eventually decrypted (see, e.g., [43, 35]). If at least one of the mixes is performed correctly, then the anonymity of votes is preserved. This approach has the benefit of tolerating ballots of arbitrary content, allowing its use with unconventional voting methods (e.g., preferential or Condorcet voting). However, it requires a complex mixing procedure; each stage of the mix must be performed by a different party (without mutual shared interest) for the scheme to be effective.

As we will show in Section 3, VoteBox employs homomorphic encryption [5] in order to keep track of each vote. A machine will encrypt a one for each candidate (or issue) the voter votes for and a zero elsewhere. The homomorphic property allow the encrypted votes for each candidate to be summed into a single total without being decrypted. This approach, also used by the Adder [30] and Civitas [12] Internet e-voting systems, typically combines the following elements:

**Homomorphic Tallying** The encryption system allows encrypted votes to be added together by a third party without knowledge of individual vote plaintexts. Many ciphers, including El Gamal public key encryption, can be designed to have this property. Anyone can verify that the final plaintext totals are consistent with the sum of the encrypted votes.

**Non-Interactive Zero Knowledge (NIZK) proofs** In any voting system, we must ensure that votes are well formed. For example, we may want to ensure that a voter has made only one selection in a race, or that the voter has not voted multiple times for the same candidate. With a plain-text ballot containing single-bit counters (i.e., 0 or 1 for each choice) this is trivial to confirm, but homomorphic counters obscure the actual counter's value with encryption. By employing NIZKs [7], a machine can include with its encrypted votes a proof that each vote is well-formed with respect to the ballot design (e.g., at most one candidate in each race received one vote, while all other candidates received zero votes). Moreover, the attached proof is *zero-knowledge* in the sense that the proof reveals no information that might help decrypt the encrypted vote. Note that although NIZKs like this can prevent a voting machine from grossly stuffing ballots, they cannot prevent a voting machine from flipping votes from one candidate to another.

**The Bulletin Board** A common feature of most cryptographic voting systems is that all votes are posted for all the world to see. Individual voters can then verify that their votes appear on the board (e.g., locating a hash value or serial number "receipt" from their voting session within a posted list of every encrypted vote). Any individual can then recompute the homomorphic tally and verify its decryption by the election authority. Any individual could likewise verify the NIZKs.

### 2.4 Non-cryptographic techniques

In response to the difficult in explaining cryptography to non-experts and as an intellectual exercise, cryptographers have designed a number of non-cryptographic paper-based voting systems that have end-to-end security properties, including ThreeBallot [39, 40], PunchScan [17], Scantegrity[1], and Prêt à Voter [10, 42]. These systems allow voters to express their vote on paper and take home a verifiable receipt. Ballots are complicated with multiple layers, scratch-off parts, or other additions to the traditional paper voting experience. A full analysis of these systems is beyond the scope of this paper.

## 3 Design

We now revisit our design goals from Section 1 and discuss their implementation in VoteBox, our complete prototype voting system.

### 3.1 User interface

**Goals achieved:** DRE-like user experience; minimized software stack

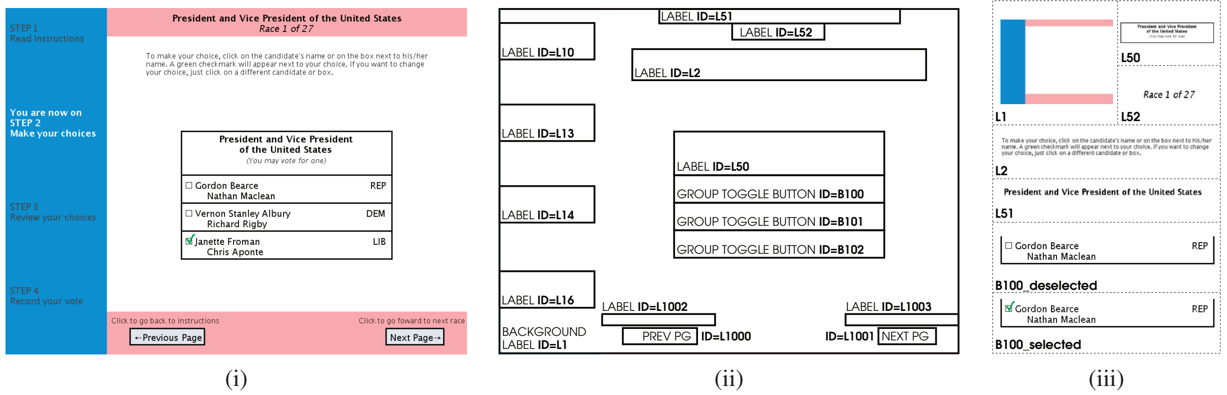
A recent study [15] bolsters much anecdotal evidence suggesting that voters strongly prefer the DRE-style electronic voting experience to more traditional methods. Cleaving to the DRE model (itself based on the archetypical computerized kiosk exemplified by bank machines, airline check-in kiosks, and the like), VoteBox presents the voter with a ballot consisting of a sequence of *pages:* full screens containing text and graphics. The only interactive elements of the interface are *buttons:* rectangular regions of the screen attached to either navigational behavior (e.g., "go to next page") or selection behavior ("choose candidate *X*"). (VoteBox supports button activation via touch screen and computer mouse, as well as keyboards and assistive technologies). An example VoteBox ballot screen is shown in Figure 1.

This simple interaction model lends itself naturally to the pre-rendered user interface, an idea popularized in the e-voting context by Yee's *Pvote* system [52, 53]. A pre-rendered ballot encapsulates both the logical content of a ballot (candidates, contests, and so forth) and the entire visual appearance down to the pixel (including all text and graphics). Generating the ballot ahead of time allows the voting machine software to perform radically fewer functions, as it is no longer required to include any code to support text rendering (including character sets, Unicode glyphs, anti-aliasing), user interface element layout (alignment, grids, sizing of elements), or any graphics rendering beyond bitmap placement.

More importantly, the entire voting machine has no need for any of these functions. The only UI-related services required by VoteBox are user input capture (in the form of $(x, y)$ pairs for taps/clicks, or keycodes for other input devies) and the ability to draw a pixmap at a given position in the framebuffer. We therefore eliminate the need for a general-purpose GUI window system, dramatically reducing the amount of code on the voting machine.

In our pre-rendered design, the ballot consists of a set of image files, a configuration file which groups these image files into pages (and specifies the layout of each page), and a configuration file which describes the abstract content of the ballot (such as candidates, races, and propositions). This effectively reduces the voting machine's user interface runtime to a state machine which behaves as follows. Initially, the runtime displays a designated initial page (which should contain instructional information and navigational components). The voter interacts with this page by selecting one of a subset of elements on the page which have been designated in the configuration as being selectable. Such actions trigger responses in VoteBox, including transitions between pages and commitment of ballot choices, as specified by the ballot's configuration files. The generality of this approach accommodates accessibility options beyond touch-screens and visual feedback; inputs such as physical buttons and sip-and-puff devices can be used to generate selection and navigation events (including "advance to next choice") for VoteBox. Audio feedback could also be added to VoteBox state transitions, again following the Pvote example [52].

We also built a ballot preparation tool to allow election administrators to create pre-rendered ballots for VoteBox. This tool, a graphical Java program, contains the layout


STEP 1
Read Instructions

You are now on
STEP 2
Make your choices

STEP 3
Review your choices

STEP 4
Record your vote

President and Vice President of the United States
*Race 1 of 27*

To make your choice, click on the candidate's name or on the box next to his/her name. A green checkmark will appear next to your choice. If you want to change your choice, just click on a different candidate or box.

President and Vice President
of the United States
*(You may vote for one)*

☐ Gordon Bearce REP
Nathan Maclean

☐ Vernon Stanley Albury DEM
Richard Rigby

✓ Janette Froman LIB
Chris Aponte

Click to go back to instructions
←Previous Page

Click to go foward to next race
Next Page→

**Figure 1: Sample VoteBox page.** The voter sees (i); a schematic for the page is shown in (ii); a subset of the pixmaps used to produce (i) are shown, along with their corresponding IDs, in (iii).

and rendering logic that is omitted from VoteBox. In addition to clear benefits that come from reducing the complexity of the voting machine, this also pushes many of the things that might change from election to election or from state to state out of the voting machine. For example, Texas requires a straight-ticket voting option while California forbids it. With VoteBox, the state-specific behavior is generated by the ballot preparation tool. This greatly simplifies the software certification process, as testing labs would only need to consider a single version of VoteBox rather than separate versions customized for each state's needs. Local groups interested in the election could then examine the local ballot definitions for correctness, without needing to trust the ballot preparation tool.

### 3.2 Auditorium

**Goals achieved:** Defense against data loss; tamper-evident audit logs

The failures described in Section 2 indicate that voting machines cannot be trusted to store their own data—or, at least, must not be *solely* trusted with their own data. We observe that modern PC equipment is sufficiently inexpensive to be used as a platform for e-voting (and note that most DREs are in fact special-purpose enclosures and extensions on exactly this sort of general-purpose hardware). VoteBox shares with recent peer-to-peer systems research the insight that modern PCs are noticeably over-provisioned for the tasks demanded of them; this is particularly true for e-voting given the extremely minimal system requirements of the user interface described in Section 3.1. Such overpowered equipment has CPU, disk, memory, and network bandwidth to spare, and VoteBox puts these to good use addressing the problem of data loss due to election-day failure.

Our design calls for all VoteBoxes in a polling place to be joined together in a broadcast network[2] as set forth in our earlier work on Auditorium [46]. An illustration of this technique can be found in Figure 2. The polling place network is not to be routable from the Internet; indeed, an air gap should exist preventing Internet packets from reaching any VoteBoxes. We will see in Section 3.3 how data *leaving* the polling place is essential to our complete design; such a one-way linkage can be built while retaining an air gap [27].

Each voting machine on the network broadcasts every event it would otherwise record in its log. As a result, the loss of a single VoteBox cannot result in the loss of its votes, or even its record of other election events. As long as a single voting machine survives, there will be some record of the votes cast that day.

**Supervisor console.** We can treat broadcast log messages as *communication* packets, with the useful side effect that these communications will be logged by all participating hosts. VoteBox utilizes this feature of Auditorium to separate machine behavior into two categories: (1) features an election official would need to use, and (2) features a voter would need to use. This dichotomy directly motivates our division of VoteBox into two software artifacts: (1) the VoteBox "booth" (that is, the voting machine component that the voter interacts with, as described in Section 3.1), and (2) the "supervisor" console.

The supervisor is responsible for the coordination of all election-day events. This includes opening the polls, closing the polls, and authorizing a vote to be captured at a booth location. For more practical reasons (because the supervisor console should run on a machine in the polling place that only election officials have physical access to, and, likewise, because election officials should never need to touch any other machine in the polling place once the election is running), this console also reports the status of every other machine in the polling place (including not

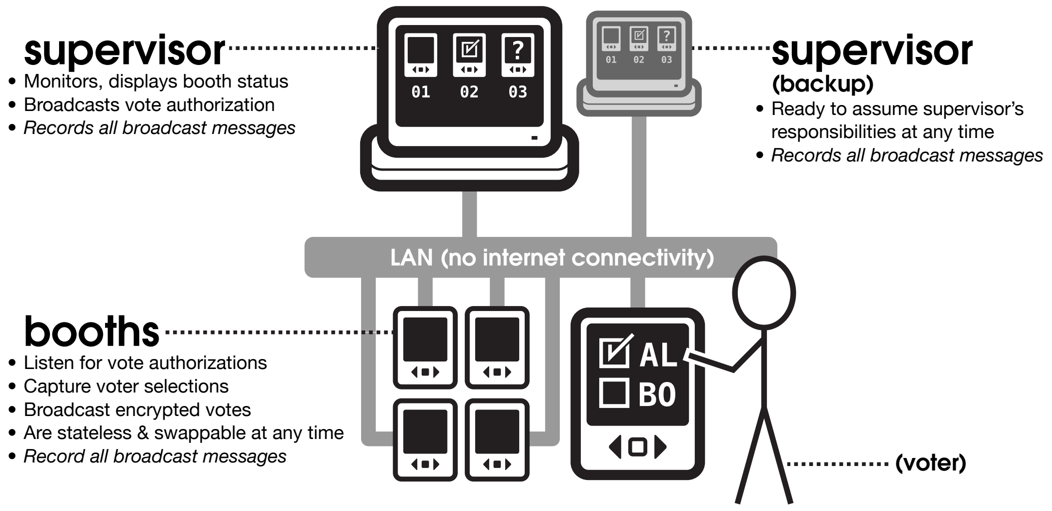

supervisor
• Monitors, displays booth status
• Broadcasts vote authorization
• Records all broadcast messages
01 02 03
supervisor
(backup)
• Ready to assume supervisor's responsibilities at any time
• Records all broadcast messages
LAN (no internet connectivity)
booths
• Listen for vote authorizations
• Capture voter selections
• Broadcast encrypted votes
• Are stateless & swappable at any time
• Record all broadcast messages
AL
BO
(voter)

**Figure 2: Voting in the Auditorium.** VoteBoxes are connected in a broadcast network. All election events (including cast ballots) are replicated to every voting machine and entangled with hash chaining. A supervisor console allows poll workers to use the Auditorium channel to distribute instructions to voting machines (such as "you are authorized to cast a ballot") such that those commands also enter the permanent, tamper-evident record.

only connectivity status, but also various "vital sign" information, such as its battery power). During the course of an election day, poll workers are able to conduct the election entirely from the supervisor console.

In addition, as an intended design decision, the separation of election control (on the supervisor console) from voting (at the VoteBox booth) fundamentally requires that every important election event be a network communication. Because we only allow this communication to happen in the form of Auditorium broadcast messages, these communications are *always* logged by *every* participating VoteBox host (supervisors and booths included).

**Hash chaining and tamper evidence.** Auditorium also provides for hash chaining of log entries; when combined with broadcast replication, the result is a lattice of hash values that entangles the timelines of individual voting machines. This technique, adapted from the field of secure audit logging [33, 48], yields strong evidence of tampering or otherwise omitted or modified records. No attacker or failure can alter any individual log entry without invalidating all subsequent hashes in the record. We prevent attackers from performing this attack in advance or arrears of the election by bookending the secure log: before the polls open, a nonce (or "launch code") is distributed, perhaps by telephone, to each polling place; this nonce is inserted into the beginning of the log. Similarly, when the polls are closed, election supervisors can quickly publish the hash of the completed log to prevent future tampering.

## 3.3 Cast ballots and immediate ballot challenge

**Goals achieved:** End-to-end verifiability

In VoteBox, cast ballots are *published* in the global Auditorium log, implicitly revealing the contents of the cast ballot to any party privy to the log data. This, of course, includes post-election auditors seeking to verify the validity and accuracy of the result, but it also could include partisans seeking proof of a bribed voter's choice (or some other sort of malicious activity). In fact, the contents of the cast ballot need to be encrypted (in order to preserve anonymity), but they also need to fit into a larger software independent design. That is, if the software (because of bugs or malice) corrupts a ballot before encrypting it, this corruption must be evident to the voter.

An *end-to-end verifiable* voting system is defined as one that can prove to the voter that (1) her vote was cast as intended and (2) her vote was counted as cast. Our design provides a challenge mechanism, which can verify the first property, along with real-time public dissemination of encrypted votes, which can satisfy the second.

**Counters.** We begin by encoding a cast ballot as an $n$-tuple of integers, each of which can be 1 or 0. Each element of the $n$-tuple represents a single choice a voter can make, $n$ is the number of choices, and a value of 1 encodes a vote *for* the choice while 0 encodes a vote *against* the choice. (In the case of propositions, both "yes" and "no" each appear as a single "choice," and in the case of candidates, each candidate is a single "choice.") The cast ballot structure needs not be organized into races or contests; it is simply an opaque list of choice values. We define each element as an integer (rather than a bit) so that ballots can be homomorphically combined. That is, ballots $A = (a_0, a_1, \ldots)$ and $B = (b_0, b_1, \ldots)$ can be summed together to produce a third ballot $S = (a_0 + b_0, a_1 + b_1, \ldots)$, whose elements are the total number of votes for each choice.[3]

**Homomorphic encryption of counters.** VoteBox uses an El Gamal variant that is additively homomorphic to encrypt ballots before they are cast. Each element of the tuple is independently encrypted. The encryption and decryption functions are defined as follows:

$$\begin{aligned} E(c, r, g^a) &= \langle g^r, (g^a)^r f^c \rangle \\ D(\langle g^r, g^{ar} f^c \rangle, a) &= \frac{g^{ar} f^c}{(g^r)^a} \\ D(\langle g^r, g^{ar} f^c \rangle, r) &= \frac{g^{ar} f^c}{(g^a)^r} \end{aligned}$$

where $f$ and $g$ are group generators, $c$ is the plaintext counter, $r$ is randomly generated at encryption time, $a$ is the decryption key, and $g^a$ is the public encryption key. To decrypt, a party needs either $a$ or $r$ in order to construct $g^{ar}$. ($g^r$, which is given as the first element of the cipher tuple, can be raised to $a$, or $g^a$, which is the public encryption key, can be raised to $r$.) After constructing $g^{ar}$, the decrypting party should divide the second element of the cipher tuple by this value, resulting in $f^c$.

To recover the counter's actual value $c$, we must invert the discrete logarithm $f^c$, which of course is difficult. As is conventional in such a situation, we accelerate this task by precomputing a reverse mapping of $f^x \rightarrow x$ for $0 < x \leq M$ (for some large $M$) so that for expected integral values of $c$ the search takes constant time. (We fall back to a linear search, starting at $M + 1$, if $c$ is not in the table.)

We now show that our encryption function is additively homomorphic by showing that when two ciphers are multiplied, their corresponding counters are added:

$$\begin{aligned} E(c_1, r_1) \odot E(c_2, r_2) &= \langle g^{r_1}, g^{ar_1} f^{c_1} \rangle \odot \langle g^{r_2}, g^{ar_2} f^{c_2} \rangle \\ &= \langle g^{r_1 + r_2}, g^{a(r_1 + r_2)} f^{c_1 + c_2} \rangle \end{aligned}$$

**Immediate ballot challenge.** To allow the voter to verify that her ballot was cast as intended, we need some way to prove to the voter that the encrypted cipher published in the Auditorium log represents the choices she *actually made*. This is, of course, a contentious issue wrought with negative human factors implications.

We term our solution to the first requirement of end-to-end verifiability "immediate ballot challenge," borrowing an idea from Benaloh [4]. A voter should be able (on any arbitrary ballot) to challenge the machine to produce a proof that the ballot was cast as intended. Of course, because these challenges generally force the voting machine to reveal information that would compromise the anonymity of the voter, challenged ballots must be discarded and not counted in the election. A malicious voting system now has no knowledge of which ballots will be challenged, so it must either cast them all correctly or risk being caught if it misbehaves.

Our implementation of this idea is as follows. Before a voter has committed to her vote, in most systems, she is presented with a final confirmation page which offers two options: (1) go back and change selections, or (2) commit the vote. Our system, like Benaloh's, adds one more page at the end, giving the voter the opportunity to challenge or cast a vote. At this point, Benaloh prints a paper commitment to the vote. VoteBox will similarly encrypt and publish the cast ballot *before* displaying this final "challenge or cast" screen. If the voter chooses to cast her vote, VoteBox simply logs this choice and behaves as one would expect, but if the voter, instead, chooses to *challenge* VoteBox, it will publish the value for $r$ that it passed to the encryption function (defined in equation 1) when it encrypted the ballot in question. Using equation 1 and this provided value of $r$, any party (including the voter) can decrypt and verify the contents of the ballot without knowing the decryption key. An illustration of this sequence of events is in Figure 3.

In order to make this process *immediate*, we need a way for voters (or voter advocates) to safely observe Auditorium traffic and capture their own copy of the log. It is only then that the voter will be able to check, in real time, that VoteBox recorded and encrypted her preferences correctly. To do this, we propose that the local network constructed at the polling place be connected to the public Internet via a data diode [27], a physical device which will guarantee that the information flow is one way. [4] This connectivity will allow any interested party to watch the polling location's Auditorium traffic in real time. In fact, any party could provide a web interface, suitable for access via smart phones, that could be used to see the voting challenges and perform the necessary cryptography. This arrangement is summarized in Figure 4. Additionally, on the output side of the data diode, we could provide a standard Ethernet hub, allowing challengers to locally plug in their own auditing equipment without relying on the election authority's network infrastructure. Because all Auditorium messages are digitally signed, there is no risk of the challenger being able to forge these messages.

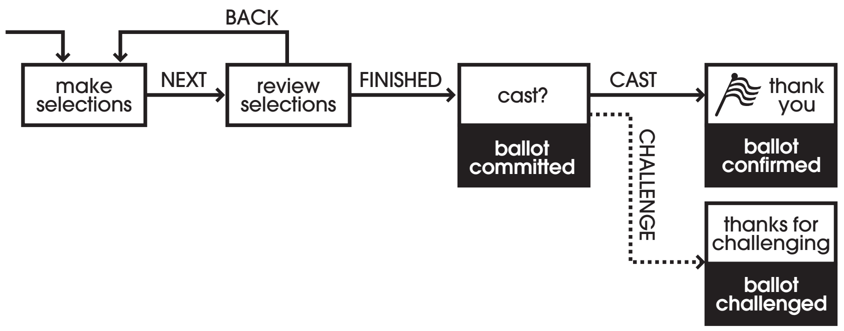


**Figure 3: Challenge flow chart.** As the voter advances past the review screen to the final confirmation screen, VoteBox commits to the state of the ballot by encrypting and publishing it. A challenger, having received this commitment (the encrypted ballot) out-of-band (see Figure 4), can now invoke the "challenge" function on the VoteBox, compelling it to reveal the contents of the same encrypted ballot. (A voter will instead simply choose "cast".)

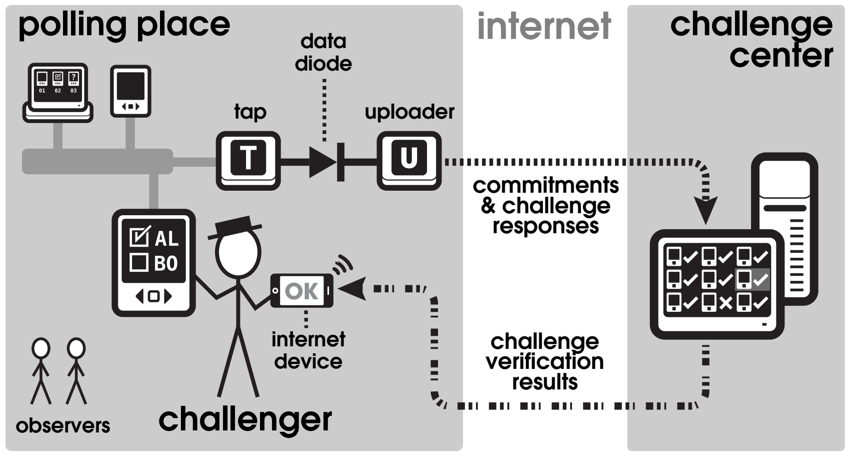


**Figure 4: Voting with ballot challenges.** The polling place from Figure 2 sends a copy of all log data over a one-way channel to election headquarters (not shown) which aggregates this data from many different precincts and republishes it. This enables third-party "challenge centers" to provide challenge verification services to the field.

**Implications of the challenge scheme.** Many states have laws against connecting voting machines or tabulation equipment to the Internet—a good idea, given the known security flaws in present equipment. Our cryptographic techniques, combined with the data diode to preserve data within the precinct, offer some mitigation against the risks of corruption in the tallying infrastructure. An observer could certainly measure the voting volume of every precinct in real-time. This is not generally considered to be private information.

VoteBox systems do not need a printer on every voting machine; however, Benaloh's printed ballot commitments offer one possibly valuable benefit: they allow any voter to take the printout home, punch the serial number into a web site, and verify the specific ballot ciphertext that belongs to them is part of the final tally, thus improving voters' confidence that their votes were counted as cast. A VoteBox lacking this printer cannot offer voters this opportunity to verify the presence of their own cast ballot ciphertexts. Challengers, of course, can verify that the ciphertexts are correctly encrypted and present in the log in real-time, thus increasing the confidence of normal voters that their votes are likewise present to be counted as cast. Optionally, Benaloh's printer mechanism could be added to VoteBox, allowing voters to take home a printed receipt specifying the ciphertext of their ballot.

Similarly, VoteBox systems do not need NIZKs. While NIZKs impose limits on the extent to which a malicious VoteBox can corrupt the election tallies by corrupting individual votes, this sort of misbehavior can be detected through our challenge mechanism. Regardless, NIZKs would integrate easily with our system and would provide an important "sanity checking" function that can apply to *every* ballot, rather than only the challenged ballots.

## 3.4 Procedures

To summarize the VoteBox design, let us review the steps involved in conducting an election with the system.

**Before the election.**

1. The ballot preparation software is used to create the necessary ballot definitions.
2. Ballot definitions are independently reviewed for correctness (so that the ballot preparation software need not be trusted).
3. Ballot definitions and key material (for vote encryption) are distributed to polling places along with VoteBox equipment.

**Election day: opening the polls.**

4. The Auditorium network is established and connected to the outside world through a data diode.
5. All supervisor consoles are powered on, connected to the Auditorium network, and one of them is enabled as the primary console (others are present for failover purposes).
6. Booth machines are powered on and connected to the Auditorium network.
7. A “launch code” is distributed to the polling place by the election administrator.
8. Poll workers open the polls by entering the launch code.

The last step results in a “polls-open” Auditorium message, which includes the launch code. All subsequent events that occur will, by virtue of hash chaining, provably have occurred *after* this “polls-open” message, which in turn means they will have provably occurred on or after election day.

**Election day: casting votes.**

9. The poll worker interacts with the supervisor console to enable a booth for the voter to use. This includes selecting a machine designated as not in use and pressing an “authorize” button.
10. The supervisor console broadcasts an authorization message directing the selected machine to interact with a voter, capture his preference, and broadcast back the result.
11. If the booth does not have a copy of the ballot definition mentioned in the authorization message, it requests that the supervisor console publish the ballot definition in a broadcast.
12. The booth graphically presents the ballot to the voter and interacts with her, capturing her choices.
13. The booth shows a review screen, listing the voter’s choices.
14. If the voter needs to make changes, she can do that by navigating backward through the ballot screens. Otherwise, she indicates she is satisfied with her selections.
15. The booth publishes the encrypted ballot over the network, thereby committing to its contents. The voter may now choose one of two paths to complete her voting session:

    **Cast her vote** by pressing a physical button. The VoteBox signals to the voter that she may exit the booth area; it also publishes a message declaring that the encrypted ballot has been officially cast and can no longer be challenged.

    **Challenge the machine** by invoking a separate UI function. The challenged VoteBox must now reveal proof that the ballot was cast correctly. It does so by publishing the secret $r$ used to encrypt the ballot; the ballot is no longer secret. This proof, like all Auditorium traffic, is relayed to the outside world, where a challenge verifier can validate against the earlier commitment and determine whether the machine was behaving correctly. The voter or poll workers can contact the challenge verifier out-of-band (e.g., with a smartphone’s web browser) to discover the result of this challenge. Finally, the ballot committed to in step 15 is nullified by the existence of the proof in the log. The VoteBox resets its state. The challenge is complete.

**Election day: closing the polls.**

16. A poll worker interacts with the supervisor console, instructing it to close the polls.
17. The supervisor console broadcasts a “polls-closed” message, which is the final message that needs to go in the global log. The hash of this message is summarized on the supervisor console.
18. Poll workers note this value and promptly distribute it outside the polling place, fixing the end of the election in time (just as the beginning was fixed by the launch code).
19. Poll workers are now free to disconnect and power off VoteBoxes.

## 3.5 Attacks on the challenge system

A key design issue we must solve is limiting communication to voters, while they are voting, that might be used to coerce them into voting in a particular fashion. If a voter could see her vote’s ciphertext before deciding to challenge it, she could be required to cast or challenge the ballot based on the ciphertext (e.g., challenge if even, cast if odd). An external observer could then catch her if she failed to vote as intended. Kelsey et al. [29] describe a variety of attacks in this fashion. Benaloh solves this problem by having the paper commitment hidden behind an opaque shield. We address it by requiring a voter to state that she intend to perform a challenge prior to approaching a voting system. At this point, a poll worker can physically lock the "cast ballot" button and enable the machine to accept a vote as normal. While the VoteBox has no idea it is being challenged, the voter (or, absolutely anybody else) can freely use the machine, videotape the screen, and observe its network behavior. The challenger cannot, however, cast the ballot.

Consequently, in the common case when voters wish to cast normal votes, *they must not have access to the* Auditorium *network stream while voting*. This means cellular phones and other such equipment must be banned to enforce the privacy of the voter. (Such a ban is already necessary, in practice, to defeat the use of cellular telephones to capture video evidence of a vote being cast on traditional DRE systems.)

A related attack concerns the behavior of a VoteBox once a user has gone beyond the "review selections" screen to the "cast?" screen (see Figure 3). If the voter wants to vote for Alice and the machine wants to defraud Alice, the machine could challenge votes for Alice while displaying the UI for a regular cast ballot. To address these phantom challenges, we take advantage of Auditorium. Challenge messages are broadcast to the entire network and initiate a suitable alarm on the supervisor console. For a genuine challenge, the supervisor will be expecting the alarm. Otherwise, the unexpected alarm would cue a supervisor to offer the voter a chance to vote again.[5] As a result, a malicious VoteBox will be unable to surreptitiously challenge legitimate votes. Rather, if it misbehaved a sufficient number of times, it would be taken out of service, limiting the amount of damage it could cause.

## 4 Discussion

### 4.1 Implementation notes and experience

Development of VoteBox has been underway since May of 2006; in that time the software has gone through a number of metamorphoses that we briefly describe here.

**Secure software design.** When we began the VoteBox implementation project, our initial goal was to develop a research platform to explore both security and human factors aspects of the electronic voting problem. Our early security approaches were: (1) reduced trusted code base through use of PRUI due to Yee [53]; (2) software simulation of hardware-enforced separation of components after the example of Sastry et al. [47]; and (3) hardware support for strict runtime software configuration control (i.e., trusted computing hardware).

Our original strategy for achieving trustworthy hardware was to target the Xbox 360 video game platform,[6] initially developing VoteBox as a Managed C# application. The Xbox has sophisticated hardware devoted to ensuring that the system runs only certified software programs, which is an obviously useful feature for a DRE. Additionally, video game systems are designed to be inexpensive and to withstand some abuse, making them good candidates for use in polling places. Finally, a lack of a sophisticated operating system is no problem for a pre-rendered user interface; we were fairly confident that an Xbox could handle displaying static pixmaps. We quickly found, however, that development for a more widely-available software platform was both easier for us and more likely to result in a usable research product.

By the end of the 2006 summer we had ported VoteBox to Java. We had no intention of relying on Java's AWT graphical interface (and its dependency, in turn, on a window system such as X or Windows). Instead, we intended to develop VoteBox atop SDL, the Simple DirectMedia Layer,[7] a dramatically simpler graphics stack. (The Pvote system also uses SDL as a side-effect of its dependency on the Pygame library [52].) Regrettably, the available Java bindings for SDL suffered from stability problems, forcing us to run our PRUI atop a limited subset of AWT (including only blitting and user input events).

Our intended approach to hardware-inspired software module separation was twofold: force all modules to interact with one another through observable software "wires," and re-start the Java VM between voters to prevent any objects lingering from one voting session to the next. Both of these ideas are due to Sastry's example. In the end, only the latter survived in our design; VoteBox essentially "reboots" between voters, but complexity and time constraints made our early software wire prototypes unworkable.

**Insecure software design.** As mentioned above, we intended from the beginning that VoteBox would serve as a foundation for e-voting research of different stripes, including human factors studies. This would prove to be its earliest test; VoteBox found use in various studies carried out by Byrne, Everett, and Greene between 2006 and 2008 [15, 16]. Working in close coordination with these researchers, we developed ballot designs and tuned the VoteBox user experience to meet their research needs. (The specific graphic design of the ballot shown in Figure 1 is owed to this collaboration.)

We also modified VoteBox to emit fine-grained data tracking the user's every move: the order of visited screens, the time taken to make choices, and so forth. This sort of functionality would be considered a breach of voter privacy in a real voting system, so we took great pains to make very clear the portions of the code that were inserted for human factors studies. Essential portions of this code were sequestered in a separate module that could be left out of compilation to ensure that no data collection can happen on a "real" VoteBox; later we made this distinction even more stark by dividing the VoteBox codebase into two branches in our source control system.

It is noteworthy that some of the most interesting human factors results [16, studies 2 and 3] require a *malicious* VoteBox. One study measured how likely voters are to notice if contests are omitted from the review screen; another, if votes on the review screen are *flipped* from the voter's actual selection. If data collection functionality accidentally left in a "real" VoteBox is bad, this code is far worse. We added the word "evil" to the names of the relevant classes and methods so that there would be no confusion in a code auditing scenario.

**S-expressions.** When it came time to develop the Auditorium network protocol, we chose to use a subset of the S-expression syntax defined by Rivest [38]. Previous experiences with peer-to-peer systems that used the convenient Java ObjectOutputStream for data serialization resulted in protocols that were awkwardly bound to particular implementation details of the code, were difficult to debug by observation of data on the wire, and were inexorably bound to Java.

S-expressions, in particular the canonical representation used in Auditorium, are a general-purpose, portable data representation designed for maximum readability while at the same time being completely unambiguous. They are therefore convenient for debugging while still being suitable for data that must be hashed or signed. By contrast, XML requires a myriad of canonicalization algorithms when used with digital signatures; we were happy to leave this large suite of functionality out of VoteBox.

We quickly found S-exps to be convenient for other portions of VoteBox. They form the disk format for our secure logs (as carbon-copies of network traffic, this is unsurprising). Pattern matching and match capture, which we added to our S-exp library initially to facilitate parsing of Auditorium messages, subsequently found heavy use at the core of Querifier [44], our secure log constraints checker, allowing its rule syntax to be naturally expressed as S-exps. Even the human factors branch of VoteBox dumps user behavior data in S-expressions.

| module | semicolons | stripped LOC |
|---|---|---|
| `sexpression` | 1170 | 2331 |
| `auditorium` | 1618 | 3440 |
| `supervisor` | 959 | 1525 |
| `votebox` | 3629 | 7339 |
| | 7376 | 14635 |

**Table 1: Size of the VoteBox trusted codebase.** *Semicolons* refers to the number of lines containing at least one ';' character and is an approximation of the number of statements in the code. *Stripped LOC* refers to the number of non-whitespace, non-comment lines of code. The difference is a crude indicator of the additional syntactic overhead of Java. Note that the ballot preparation tool is not considered part of the TCB, since it generates ballots that should be audited directly; it is 4029 semicolons (6657 stripped lines) of Java code using AWT/Swing graphics.

**Code size.** Table 1 lists several code size metrics for the modules in VoteBox, including all unit tests. We aspired to the compactness of Pvote's 460 Python source lines [52], but the expanded functionality of our system, combined with the verbosity of Java (especially when written in clear, modern object-oriented style) resulted in a much larger code base. The `votebox` module (analogous to Pvote's functionality) contains nearly twenty times as many lines of code. The complete VoteBox codebase, however, compares quite favorably with current DRE systems, making thorough inspection of the source code a tractable proposition.

## 4.2 Performance evaluation and estimates

By building a prototype implementation of our design, we are able to validate that it operates within reasonable time and space bounds. Some aspects of VoteBox require "real time" operation while others can safely take minutes or hours to complete.

**Log publication.** Recall that VoteBoxes, by virtue of the fact that they communicate with one another using the Auditorium protocol, produce s-expression log data which serves as a representation of the events that happened during the election. An important design goal is the allowance of outside parties to see this log data in real time; our immediate ballot challenge protocol relies on it.

We've assumed, as a worst case, that the polling place is connected to election central with a traditional modem. This practical bandwidth limitation forces us to explore the size of the relevant log messages and examine their impact on the time it takes to perform an immediate ballot challenge. This problem is only relevant if the verification machine is not placed on the polling place network (on the public side of the data diode). With the verification machine on the LAN, standard network technology will be able to transmit the log data much faster than any reasonable polling place could generate it.

A single voter's interaction with the polling place results in in the following messages: (1) an authorization message from the supervisor to the booth shortly after the voter enters the polling place, (2) a commitment message broadcast by the booth after the voter is done voting, (3) either a cast ballot message or a challenge response message (the former if the voter decides to cast and the latter if the voter decides to challenge), (4) and an acknowledgment from the supervisor that the cast ballot or challenge has been received, which effectively allows the machine to release its state and wait for the next authorization.

Assuming all the crypto keys are 1024-bits long, an authorization-to-cast message is 1 KB. Assuming 30 selectable elements are on the ballot, both commit and cast messages are 13 KB while challenge response messages are 7 KB. An acknowledgment is 1 KB.

We expect a good modem's throughput to be 5 KB/second. The challenger must ask the machine to commit to a vote, wait for the verification host to receive the commitment, then ask the machine to challenge the vote. (The voter *must* wait for proof of the booth's commitment in order for the protocol to work.) In the best case, when only one voter is in the polling place (and the uploader's buffer is empty), a commitment can be immediately transmitted. This takes under 3 seconds. The challenge response can be transmitted in under 2 seconds. In the worst case, when as many as 19 other voters have asked their respective booths to commit and cast their ballots, the challenger must wait for approximately 494 KB of data to be uploaded (on behalf of the other voters). This would take approximately 100 seconds. Assuming 19 additional voters, in this short time, were given access to booths and all completed their ballots, the challenger might be forced to wait another 100 seconds before the challenge response (the list of $r$-values used to encrypt the first commitment) could make it through the queue.

Therefore, in the absolute worst case situation (30 elements on the ballot and 20 machines in the polling place), the challenger is delayed by a maximum of 200 seconds due to bandwidth limitations.

**Encryption.** Because a commitment is an encrypted version of the cast ballot, a cast ballot must be encrypted before a commitment to it is published. Furthermore, the verifier must do a decryption in order to verify the result of a challenge. Encryption and decryption are always a potential source of delay, therefore we examine our implementation's encryption performance here.

Recall that a cast ballot is an $n$-tuple of integers, and an encrypted cast ballot has each of these integers encrypted using our additively homomorphic El Gamal encryption function. We benchmarked the encryption of a reference 30 candidate ballot; on a Pentium M 1.8 GHz laptop it took 10.29 CPU seconds, and on an Opteron 2.6 GHz server it took 2.34 CPU seconds. We also benchmarked the decryption, using the $r$-values generated by the encryption function (simulating the work of a verification machine in the immediate ballot challenge protocol). On the laptop, this decryption took 5.18 CPU seconds, and on the server it took 1.27 CPU seconds.

The runtime of this encryption and decryption will be roughly the same. However, there is one caveat. To make our encryption function *additively* homomorphic, we exponentiate a group member (called $f$ in equation 1) by the plaintext counter (called $c$ in equation 1). (The result is that when this value is multiplied, the original counter gets added "in the exponent.") Because discrete log is a hard problem, this exponentiation cannot be reversed. Instead, our implementation stores a precomputed table of encryptions of low counter values. We assumes that, in real elections, these counters will never be above some reasonable threshold (we chose 20,000). Supporting counters larger than our precomputed table would require a very expensive search for the proper value.

This is never an issue in practice, since individual ballots only ever encrypt the values 0 and 1, and there will never be more than a few thousand votes per day in a given precinct. While there may be a substantially larger number of votes across a large city, the election official only needs to perform the homomorphic addition and decryption on a precinct-by-precinct basis.[8] This also allows election officials to derive per-precinct subtotals, which are customarily reported today and are not considered to violate voter privacy. Final election-night tallies are computed by adding the plaintext sums from each precinct.

**Log analysis.** There are many properties of the published logs that we might wish to validate, such as ensuring that all votes were cast while the polls were open, that no vote is cast without a prior authorization sharing the same nonce, and so on. These properties can be validated by hand, but are also amenable to automatic analysis. We built a tool called QUERIFIER [44, 45] that performs this function based on logical predicates expressed over the logs. None of these queries need to be validated in real time, so performance is less critical, so long as answers are available within hours or even days after the election.

## 4.3 Security discussion

Beyond the security goals introduced in Section 1 and elaborated in Section 3, we offer a few further explorations of the security properties of our design.

**Ballot decryption key material.** We have thus far avoided the topic of which parties are entitled to decrypt the finished tally, assuming that there exists a single entity (perhaps the director of elections) holding an El Gamal private key. We can instead break the decryption key up into shares [49, 13] and distribute them to several mutually-untrusting individuals, such as representatives of each major political party, forcing them to cooperate to view the final totals.

This may be insufficient to accommodate varying legal requirements. Some jurisdictions require that each county, or even each polling place, be able to generate its own tallies on the spot once the polls close. In this case we must create separate key material for each tallying party, complicating the matter of who should hold the decryption key. Our design frees us to place the decryption key on, e.g., the supervisor console, or a USB key held by a local election administrator. We can also use threshold decryption to distribute key shares among multiple VoteBoxes in the polling place or among mutually-untrusting individuals present in the polling place.

**Randomness.** Our El Gamal-based cryptosystem, like many others, relies on the generation of random numbers as part of the encryption process. Since the ciphertext includes $g^r$, a malicious voting machine could perform $O(2^k)$ computations to encode $k$ bits in $g^r$, perhaps leaking information about voters' selections. Karlof et al. [28] suggest several possible solutions, including the use of trusted hardware. Verifiable randomness may also be possible as a network service or a multi-party computation within the VoteBox network [21].

**Mega attacks.** We believe the Auditorium network offers defense against mishaps and failures of the sort already known to have occurred in real elections. We further expect the networked architecture to provide some defense against more extreme failures and attacks that are hypothetical in nature but nonetheless quite serious. These "mega attacks," such as post-facto switched results, election-day shadow polling places, and armed booth capture (described more fully in previous work [46]), are challenges for any electronic voting system (and even most older voting technologies as well).

## 5 Conclusions and future work

In this paper we have shown how the VoteBox system design is a response to threats, real and hypothesized, against the trustworthiness of electronic voting. Recognizing that voters prefer a DRE-style system, we endeavored to create a software platform for e-voting projects and then assembled a complete system using techniques and ideas from current research in the field. VoteBox creates audit logs that are believable in the event of a post-facto audit, and it does this using the Auditorium networking layer, allowing for convenient administration of polls as well as redundancy in case of failure. Its code complexity is kept under control by moving inessential graphics code outside the trusted system, with the side effect that ballot descriptions can be created—and audited—long before election day. Finally, the immediate ballot capture technique gives real power to random machine audits. Any voter can ask to challenge any voting machine, and the machine has no way to know it is under test before it commits to the contents of the encrypted ballot.

VoteBox is a complete system and yet still an ongoing effort. It is still being actively used for human factors experimentation, work which spurs evolution and maturity of the software. Many of VoteBox's features were designed with human factors of both poll workers and voters in mind. Evaluating these with human subject testing would make a fascinating study. For example, we could evaluate the rate at which voters accidentally challenge ballots, or we could ask voters to become challengers and see if they can correctly catch a faulty machine.

We have a number of additional features and improvements we intend to add or are in the process of adding to the system as well. Because one of the chief benefits of the DRE is its accessibility potential, we anticipate adding support for unusual input devices; similarly, following the example of Pvote, we expect that VoteBox's ballot state machines will map naturally onto the problem of providing a complete audio feedback experience to match the video display. As we continue to support human factors testing, it is obviously of interest to continue to maintain a clear separation and identification of "evil" code; techniques to statically determine whether this code (or other malicious code) is present in VoteBox will increase our assurance in the system. We are in the process of integrating NIZK proofs into our El Gamal encrypted vote counters, further bolstering out assurance that VoteBox systems are behaving correctly. We intend to expand our use of Querifier to automatically and conveniently analyze Auditorium logs and confirm that they represent valid election events. A tabulation system for VoteBox is another logical addition to the architecture, completing the entire election life cycle from ballot design through election-day voting (and testing) to post-election auditing and vote tabulation. Finally, we note that as a successful story of combining complementary e-voting research advances, we are on the lookout for other suitable techniques to include in the infrastructure to further enhance the end-to-end verifiability, in hope of approaching true software independence in a voter-acceptable way.

## Acknowledgments

This work was funded in part by NSF grants CNS-0524211 and CNS-0509297. Portions of this work took place during Wallach's sabbatical at Stanford and SRI and during Derr's internship at SRI; we thank those institutions for their support. We also thank Ben Adida, Josh Benaloh, Peter Neumann, Chris Piekert, and Brent Waters for many helpful discussions on the VoteBox architecture. In addition to the authors of this paper, the following Rice students have contributed to the VoteBox codebase: Emily Fortuna, George Mastrogiannis, Kevin Montrose, Corey Shaw, and Ted Torous. We also acknowledge Mike Byrne, Sarah Everett, and Kristen Greene for designing VoteBox's ballots. Finally, we thank the anonymous referees for their helpful and detailed feedback.

## Notes

[1] http://www.scantegrity.org

[2] The Hart InterCivic eSlate voting system also includes a polling place network and is superficially similar to our design; unfortunately, the eSlate system has a variety of security flaws [24] and lacks the fault tolerance, auditability, and end-to-end guarantees provided by VoteBox.

[3] While this simple counter-based ballot does not accommodate write-in votes, homomorphic schemes exist that allow more flexible ballot designs, including write-ins [31].

[4] An interesting risk with a data diode is ensuring that it is installed properly. Polling place systems could attempt to ping known Internet hosts or otherwise map the local network topology, complaining if two-way connectivity can be established. We could also imagine color-coding cables and plugs to clarify how they must be connected.

[5] Invariably, some percentage of regular voters will accidentally challenge their ballots. By networking the voting machines together and raising an alarm for the supervisor, these accidental challenges will only inconvenience these voters rather than disenfranchising them. Furthermore, accidental challenges helpfully increase the odds of machines being challenged, making it more difficult for a malicious VoteBox to know when it might be able to cheat.

[6] The VoteBox name derives in part from this early direction, known at the time as the "ballotbox 360".

[7] http://www.sdl.org

[8] Vote centers, used in some states for early voting and others for election day, will have larger numbers of votes cast than traditional small precincts. Voting machines could be grouped into subsets that would have separate Auditorium networks and separate homomorphic tallies. Similarly, over a multi-day early voting period, each day could be treated distinctly.

## References

[1] Final Report of the Cuyahoga Election Review Panel, July 2006. http://cuyahogavoting.org/CERP_Final_Report_20060720.pdf.

[2] A. Ash and J. Lamperti. Florida 2006: Can statistics tell us who won Congressional District-13? *Chance*, 21(2), Spring 2008.

[3] N. Barkakati. *Results of GAO's Testing of Voting Systems Used in Sarasota County in Florida's 13th Congressional District*. Government Accountability Office, Feb. 2008. Report number GAO-08-425T.

[4] J. Benaloh. Ballot casting assurance via voter-initiated poll station auditing. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, Aug. 2007.

[5] J. D. C. Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University Department of Computer Science, 1987.

[6] M. Blaze, A. Cordero, S. Engle, C. Karlof, N. Sastry, M. Sherr, T. Stegers, and K.-P. Yee. *Source Code Review of the Sequoia Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/sequoia-source-public-jul26.pdf.

[7] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112, New York, NY, USA, 1988.

[8] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. P. Zeller. *Source Code Review of the Diebold Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/diebold-source-public-jul29.pdf.

[9] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[10] D. Chaum, P. Y. A. Ryan, and S. A. Schneider. A practical, voter-verifiable election scheme. In *ESORICS '05*, pages 118–139, Milan, Italy, 2005.

[11] M. Cherney. *Vote results further delayed*. The Sun News, Jan. 2008. http://www.myrtlebeachonline.com/news/local/story/321972.html.

[12] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[13] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 307–315, Santa Barbara, CA, July 1989.

[14] D. L. Dill and D. S. Wallach. *Stones Unturned: Gaps in the Investigation of Sarasota's Disputed Congressional Election*, Apr. 2007. http://www.cs.rice.edu/~dwallach/pub/sarasota07.html.

[15] S. Everett, K. Greene, M. Byrne, D. Wallach, K. Derr, D. Sandler, and T. Torous. Is newer always better? The usability of electronic voting machines versus traditional methods. In *Proceedings of CHI 2008*, Florence, Italy, Apr. 2008.

[16] S. P. Everett. *The Usability of Electronic Voting Machines and How Votes Can Be Changed Without Detection*. PhD thesis, Rice University, Houston, TX, 2007.

[17] K. Fisher, R. Carback, and T. Sherman. Punchscan: Introduction and system definition of a high-integrity election system. In *Workshop On Trustworthy Elections (WOTE 2006)*, Cambridge, U.K., June 2006.

[18] Florida Department of State, Division of Elections, Tallahassee, Florida. *Parallel Test Summary Report*, Dec. 2006. http://election.dos.state.fl.us/pdf/parallelTestSumReprt12-18-06.pdf.

[19] Florida Department of State, Division of Elections, Tallahassee, Florida. *Audit Report of the Election Systems and Software, Inc's, iVotronic Voting System in the 2006 General Election for Sarasota County, Florida*, Feb. 2007. http://election.dos.state.fl.us/pdf/auditReportSarasota.pdf.

[20] L. Frisina, M. C. Herron, J. Honaker, and J. B. Lewis. *Ballot Formats, Touchscreens, and Undervotes: A Study of the 2006 Midterm Elections in Florida*. Dartmouth College and The University of California at Los Angeles, May 2007. Originally released Nov. 2006, current draft available at http://www.dartmouth.edu/~herron/cd13.pdf.

[21] R. Gardner, S. Garera, and A. D. Rubin. Protecting against privacy compromise and ballot stuffing by eliminating non-determinism from end-to-end voting schemes. Technical Report 245631, Johns Hopkins University, Apr. 2008. http://www.cs.jhu.edu/~ryan/voting_randomness/ggr_voting_randomness.pdf.

[22] S. N. Goggin and M. D. Byrne. An examination of the auditability of voter verified paper audit trail (VVPAT) ballots. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Berkeley, CA, USA, Aug. 2007.

[23] S. Hertzberg. *DRE Analysis for May 2006 Primary, Cuyahoga County, Ohio*. Election Science Institute, San Francisco, CA, Aug. 2006. http://bocc.cuyahogacounty.us/GSC/pdf/esi_cuyahoga_final.pdf.

[24] S. Inguva, E. Rescorla, H. Shacham, and D. S. Wallach. *Source Code Review of the Hart InterCivic Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/Hart-source-public.pdf.

[25] D. Jefferson. What happened in Sarasota County? *The Bridge (National Academy of Engineering)*, 37(2), Summer 2007. Also available online at http://www.nae.edu/nae/bridgecom.nsf/weblinks/MKEZ-744KWK?OpenDocument.

[26] D. W. Jones. Parallel testing during an election, 2004. http://www.cs.uiowa.edu/~jones/voting/testing.shtml#parallel.

[27] D. W. Jones and T. C. Bowersox. Secure data export and auditing using data diodes. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Vancouver, B.C., Canada, Aug. 2006.

[28] C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *USENIX Security Symposium*, Aug. 2005.

[29] J. Kelsey, A. Regenscheid, T. Moran, and D. Chaum. Hacking paper: Some random attacks on paper-based E2E systems. Presentation in Seminar 07311: Frontiers of Electronic Voting, 29.07.07–03.08.07, organized in The International Conference and Research Center for Computer Science (IBFI, Schloss Dagstuhl, Germany), Aug. 2007. http://kathrin.dagstuhl.de/files/Materials/07/07311/07311.KelseyJohn.Slides.pdf.

[30] A. Kiayias, M. Korman, and D. Walluck. An Internet voting system supporting user privacy. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 165–174, Washington, DC, USA, 2006.

[31] A. Kiayias and M. Yung. The vector-ballot e-voting approach. In *FC'04: Financial Cryptography 2004*, Key West, FL, Feb. 2004.

[32] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *Proc. of IEEE Symposium on Security & Privacy*, Oakland, CA, 2004.

[33] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[34] W. R. Mebane and D. L. Dill. *Factors Associated with the Excessive CD-13 Undervote in the 2006 General Election in Sarasota County, Florida*. Cornell University and Stanford University, Jan. 2007. http://macht.arts.cornell.edu/wrm1/smachines1.pdf.

[35] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *CCS '01: Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 116–125, Philadelphia, PA, 2001.

[36] S. Pynchon and K. Garber. *Sarasota's Vanished Votes: An Investigation into the Cause of Uncounted Votes in the 2006 Congressional District 13 Race in Sarasota County, Florida*. Florida Fair Elections Center, DeLand, Florida, Jan. 2008. http://www.floridafairelections.org/reports/Vanishing_Votes.pdf.

[37] D. Rather. The trouble with touch screens. Broadcast on HDNet, also available at http://www.hd.net/drr227.html, Aug. 2007.

[38] R. L. Rivest. S-expressions. IETF Internet Draft, May 1997. http://people.csail.mit.edu/rivest/sexp.txt.

[39] R. L. Rivest. The ThreeBallot voting system. http://theory.csail.mit.edu/~rivest/Rivest-TheThreeBallotVotingSystem.pdf, Oct. 2006.

[40] R. L. Rivest and W. D. Smith. Three voting protocols: ThreeBallot, VAV, and Twin. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, Aug. 2007.

[41] R. L. Rivest and J. P. Wack. On the notion of "software independence" in voting systems, 2006. http://vote.nist.gov/SI-in-voting.pdf.

[42] P. Y. A. Ryan and T. Peacock. A threat analysis of Prêt à Voter. In *Workshop On Trustworthy Elections (WOTE 2006)*, Cambridge, U.K., June 2006.

[43] K. Sako and J. Kilian. Receipt-free mix-type voting scheme - a pracitcal solution to the implementation of a voting booth. In *Advances in Cryptology: EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer-Verlag, 1995.

[44] D. Sandler, K. Derr, S. Crosby, and D. S. Wallach. Finding the evidence in tamper-evident logs. Technical Report TR08-01, Department of Computer Science, Rice University, Houston, TX, Jan. 2008. http://cohesion.rice.edu/engineering/computerscience/TR/TR_Download.cfm?SDID=238.

[45] D. Sandler, K. Derr, S. Crosby, and D. S. Wallach. Finding the evidence in tamper-evident logs. In *Proceedings of the 3rd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'08)*, Oakland, CA, May 2008.

[46] D. Sandler and D. S. Wallach. Casting votes in the Auditorium. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, Aug. 2007.

[47] N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, Aug. 2006.

[48] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.

[49] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[50] C. Stewart, III. Plaintiff Jennings Ex. 8. (for Dec. 19, 2006 evidentiary hr'g), Jennings v. Elections Canvassing Comm'n of the State of Florida et al., No. 2006 CA 2973 (Circuit Ct. of the 2d Judicial Circuit, Leon County, Fla., filed Nov. 20, 2006), reproduced in 2 Appendix to Emergency Petition for a Writ of Certiorari A-579-80, Jennings v. Elections Canvassing Comm'n of the State of Florida et al., No. 1D07-11 (Fla. 1st Dist. Ct. of Appeal, filed Jan. 3, 2007), Dec. 2006.

[51] A. Yasinsac, D. Wagner, M. Bishop, T. Baker, B. de Medeiros, G. Tyson, M. Shamos, and M. Burmester. *Software Review and Security Analysis of the ES&S iVotronic 8.0.1.2 Voting Machine Firmware*. Security and Assurance in Information Technology Laboratory, Florida State University, Tallahassee, Florida, Feb. 2007. http://election.dos.state.fl.us/pdf/FinalAudRepSAIT.pdf.

[52] K.-P. Yee. Extending prerendered-interface voting software to support accessibility and other ballot features. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, Aug. 2007.

[53] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *USENIX/ACCURATE Electronic Voting Technology Workshop*, Vancouver, B.C., Canada, 2006.

# An Empirical Security Study of the Native Code in the JDK

Gang Tan and Jason Croft
*Boston College*
*gtan@cs.bc.edu, croftj@bc.edu*

## Abstract

It is well known that the use of native methods in Java defeats Java's guarantees of safety and security, which is why the default policy of Java applets, for example, does not allow loading non-local native code. However, there is already a large amount of trusted native C/C++ code that comprises a significant portion of the Java Development Kit (JDK). We have carried out an empirical security study on a portion of the native code in Sun's JDK 1.6. By applying static analysis tools and manual inspection, we have identified in this security-critical code previously undiscovered bugs. Based on our study, we describe a taxonomy to classify bugs. Our taxonomy provides guidance to construction of automated and accurate bug-finding tools. We also suggest systematic remedies that can mediate the threats posed by the native code.

## 1 Introduction

Since its birth in the mid 90s, Java has grown to be one of the most popular computing platforms. Recognizing Java's importance, security researchers have scrutinized Java's security from its early days (c.f., [8, 28, 32, 25]). Various vulnerabilities in the Java security model have been identified and fixed; formal models of various aspects of Java security have been proposed (e.g., [41, 13]), sometimes with machine-checked theorems and proofs [22].

In this paper we examine a less-scrutinized aspect of Java security: the native methods used by Java classes. It is well known that once a Java application uses native C/C++ methods through the Java Native Interface (JNI), any security guarantees provided by Java might be invalidated by the native methods. Figure 1 shows a contrived example. The Java class "Vulnerable" contains a native method, which is realized by a C function. The C function is susceptible to a buffer overflow as it performs an unbounded string copy to a 512-byte buffer. Consequently, an attacker can craft malicious inputs to the public Java byteCopy() method, and overtake the JVM.

Java code

```
class Vulnerable {
  //Declare a native method
  private native void bcopy(byte[] arr);
  public void byteCopy(byte[] arr) {
    //Call the native method
    bcopy(arr);
  }
  static {
    System.loadLibrary("Vulnerable");
  }
}
```

C code

```
#include <jni.h>
#include "Vulnerable.h"
JNIEXPORT void JNICALL Java_Vulnerable_bcopy
  (JNIEnv *env, jobject obj, jobject arr)
{
  char buffer[512];
  jbyte *carr;
  carr = (*env)->GetByteArrayElements
                  (env,arr,0);
  //Unbounded string copy to a local buffer
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements
            (env,arr,carr,0);
}
```

Figure 1: Vulnerable JNI Code.

Due to the fundamental insecurity of native C/C++ code, the default policy of Java applets, for example, does not allow loading non-local native code. Nonetheless, there is already a large amount of trusted native code that comprises a significant portion of the Java Development Kit (JDK). For instance, the classes under java.util.zip in Sun's JDK are just wrappers that invoke the popular Zlib C library. In JDK 1.6, there are over 800,000 lines of C/C++ code. Over the time, the size of C/C++ code has been on the increase: JDK 1.4.2 has 500,000 lines; JDK 1.5 has 700,000 lines; and JDK 1.6 has 800,000 lines. Any vulnerability in this trusted native code can compromise the security of the JVM. Several vulnerabilities have already been discovered in this code [33, 38, 37].

Since the native code in the JDK is critical to Java security, examining and ensuring its security is of great practical value. As a first step toward this goal, we have carried out an empirical security study of this large and security-critical code. Our research makes the following contributions:

- This is the first systematic security study of the native code in Sun's JDK, a security-critical and ubiquitous piece of software. A few sporadic bug reports exist, but none have scrutinized this aspect of Java security.
- We discovered previously unknown security-critical bugs (59 in total). By removing them, the overall Java security will be strengthened. Furthermore, we describe a taxonomy of bugs based on our study (Section 3). New bug patterns that arise in the context of the JNI are discussed and analyzed. Our taxonomy provides guidance to construction of scalable and accurate bug-finding tools.
- We will propose remedies (Section 4) to mediate the threats posed by the native code, with various tradeoffs among security, performance, and effort. We also discuss limitations of current approaches and point out future directions.

## 2 Overview of the JDK's native code and our approach to characterizing bug patterns

The JNI is Java's mechanism for interfacing with native C/C++ code. Programmers use the `native` modifier to declare native methods in Java classes (e.g. the `bcopy` method in Figure 1 is declared as a native method). Once declared, native methods can be invoked in Java in the same way as how ordinary Java methods are invoked. Programmers then provide in C or C++ implementation of the declared native methods. The implementation can use various API functions provided by the JNI interface to cooperate with the Java side. Through the API functions, native methods can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on.

In the source directories `share/native`, `solaris/native`, and `windows/native` of Sun's JDK 1.6 (v6u2), there are over 800,000 lines of C/C++ code (counted using `wc`). The native code in these directories implements the native methods declared in the JDK classes. The native code in the directory `share/native` is shared across platforms, while the code in `solaris/native` and `windows/native` is platform dependent. The majority of the native code in the JDK is in the C language; around 700,000 lines are in C, while the rest are in C++. In our following discussion, we will mostly refer to the C code in the JDK. All of our discussion, unless specially noted, applies to the C++ code as well.

The 800k lines of native code can be conceptually divided into two parts: *library code* and *interface code*. The library code is the C code that belongs to a common C library. For example, the code under `share/native/java/util/zip/zlib-1.1.3` is from Zlib 1.1.3. The interface code implements Java native methods, and glues Java with C libraries through the JNI. For example, the C code in `native/java/util/zip/Deflater.c` implements the native methods in the `java.util.zip.Deflater` class, and glues Java with the Zlib C library.

**Our approach to characterizing bug patterns.** Given the large amount of trusted native code in the JDK, bugs are likely to exist. Our ultimate goal is to build highly automatic tools that can identify bugs in the JDK's native code. However, as no general methodology exists to identify all bugs accurately in a program, we believe that the important first step is to collect empirical evidence, and characterize relevant bug patterns. Only after this due diligence, we can select the right techniques to take advantage of the domain knowledge of the JDK and the JNI, and construct effective bug-finding tools.

In the first step, we intend to cover as many bug patterns as we can. We decided to scan the source code using off-the-shelf static analysis tools and also simple tools (scripts and scanners) built by us. Although these tools are inaccurate, their scanning results are fairly complete and thus enable us to compile enough evidence to conclude the characteristics of bug patterns. Next, we discuss the tools used in our study:

- To scan the common bug patterns inside C code, such as buffer overflows, integer overflows, and race conditions, we used a combination of Splint [10], Cigital's ITS4 [39], and Flawfinder [42]. We chose a combination of these tools, rather than a single one, because their strengths complement one another. For example, Splint performs full parsing and can flag many incompatible type casts. ITS4 and Flawfinder can flag time-of-check-to-time-of-use (TOCTTOU) flaws, among others.
- Some bug patterns in the JDK's native code are

particular to the Java Native Interface (JNI) and we cannot use existing tools to scan for errors in these patterns. We have built simple tools, including grep-based scripts and scanners implemented in CIL [30], to search for bugs in these patterns.

- For the list of warnings produced by the static analysis tools, we manually inspected the source code to identify true bugs. To help the manual inspection, we used the GNU GLOBAL Source Code Tag System [16] to build a database of tags in the JDK source code, and used htags to generate HTML files for the source code. This made source-code navigation much easier. For example, with one click, we can find all places where a particular function is invoked.

Although the foregoing approach is sufficient for characterizing bug patterns, it is clear the tools will not be scalable to cover all 800,000 lines of native code in the JDK. In Section 4, we will discuss techniques that make a significant progress toward providing safety to the JDK's native code.

**Target directories.** Limited by our time to perform manual inspection, we focused our study on the code under the directories `share/native/java` and `solaris/native/java`. We will call these directories the target directories in the following text. The target directories include approximately 38,000 lines of C code, which implement the native methods in the `java.*` classes.

## 3 Taxonomy of bugs in the JDK's native code

We now present a collection of bug patterns in the JDK's native code. Some of these patterns are well known, such as buffer overflows, but we will discuss them in the context of the JDK. Some bug patterns are due to the mismatch between Java's programming model and C's, and thus are unique in the context.

Table 1 shows a summary of the results of our security study. For each bug pattern, the table shows the number of bugs we identified. We include a bug in the table when two conditions hold. First, there must be a *programming error* in the native code. For example, the C code in Figure 1 has a programming error, which performs an unbounded string copy. The second condition is that an attacker must be able to trigger the programming error. For the example in Figure 1, the attacker can trigger the error of unbounded string copy by passing malicious data to the `Vulnerable` class.

Table 1 also classifies whether a bug pattern is *security critical*. We define that a bug pattern is security critical if, by exploiting bugs in the pattern, an attacker can take over the JVM, gain authorized privileges, or crash the JVM (a denial-of-service attack). A security-critical bug is a vulnerability.

Finally, Table 1 shows the static analysis tools we used to identify the bugs in a bug pattern, and the section that describes detailed findings on the bug pattern. In each section, we will show representative examples, but refer readers to the appendix of our technical report [35] for a full list of the bugs we identified. We will also suggest ad-hoc fixes for some bug patterns, but defer discussions of more systematic remedies to the next section.

Not included in the table are the false positive rates of the static analysis tools; they will be presented when we discuss static analysis as a remedy in the next section.

### 3.1 Unexpected control flows due to mishandling Exceptions

The JNI interface provides API functions such as `Throw` and `ThrowNew` for raising Java exceptions. By throwing an exception, a native method can notify the JVM of errors. However, there is a mismatch between Java's exception-handling mechanism and the JNI's. In Java, when an exception occurs, the JVM automatically transfers the control to the nearest enclosing try/catch statement that matches the exception type. In contrast, an exception raised through the JNI does not immediately disrupt the native method execution, and only after the native method finishes execution will the JVM mechanism for exceptions start to take over. Therefore, JNI programmers must explicitly implement the control flow after an exception has occurred, by either immediately returning to Java or checking and clearing the exception explicitly using JNI API functions such as `ExceptionOccurred` and `ExceptionClear`.

Because Java and the JNI handle exceptions differently, it is easy for JNI programmers to make mistakes. Figure 2 presents a contrived example that shows how mishandling of exceptions may lead to vulnerabilities. At first sight, the `strcpy` from the incoming Java array to a local buffer is safe: there is a bounds check before the copy, and when the check fails, an exception is thrown. However, since the exception does not disrupt the control flow, the `strcpy` will always be executed and may result in an unbounded string copy. This example shows that mishandling exceptions creates unexpected control-flow paths where dangerous operations might happen.

The fix for the example in Figure 2 is simple—just put a return statement after the throwing-exception statement. However, it becomes complicated when function

| Bug patterns | | Errors | Security Critical | Static Tools Used | Section |
|---|---|---|---|---|---|
| Unexpected control flows due to mishandling exceptions | | 11 | Y | grep-based scripts | 3.1 |
| C pointers as Java integers | | 38 | N | Our scanner (implemented in CIL) | 3.2 |
| Race conditions in file accesses | | 3 | Y | ITS4, Flawfinder | 3.3 |
| Buffer overflows | | 5* | Y | Splint, ITS4, Flawfinder | 3.4 |
| Mem. management flaws | C mem. | 1 | N | Splint | 3.5 |
| | Java mem. | 28 | N | grep-based scripts | |
| Insufficient error checking | JNI APIs | 35 | Y | grep-based scripts | 3.6 |
| | misc. | 5 | Y | Splint | |
| Total | | 126 | 59 | | |

*One buffer-overflow flaw is not in the target directory.

Table 1: A summary of the bugs we identified in the target directories.

```
void Java_Vulnerable_bcopy (JNIEnv *env, jobject obj, jbyteArray jarr) {
  char buffer[512];

  if ((*env)->GetArrayLength(env, jarr) > 512) {
    JNU_ThrowArrayIndexOutOfBoundsException(env, 0);
  }

  //Get a pointer to the Java array, then copy the Java array to a local buffer
  jbyte *carr = (*env)->GetByteArrayElements(env, jarr, NULL);
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements(env,arr,carr,0);
}
```

Figure 2: An example of mishandling JNI exceptions

calls are involved. Imagine a C function, say `f`, invokes another C function, say `g`, and the function `g` throws an exception when an error occurs. The `f` function has to explicitly deal with two cases of calling `g`: the successful case, and the exceptional case. Mishandling it may result in the same error as the one in Figure 2. It becomes much more complicated when the C function `f` invokes a Java method. The JVM mechanism for exceptions will not take effect until the C function returns, even for the exceptions raised in the Java method.

We developed a grep-based script to search for all places where an exception is explicitly thrown. Of the 337 hits in the target directories, we found 11 places where the control flows for exceptions are implemented incorrectly. A representative example from `solaris/native/java/lang/UNIXProcess_md.c` is shown in Figure 3. The macro `NEW` invokes the function `xmalloc`, which in turn invokes `malloc` to allocate a specified amount of memory. If the `malloc` function returns null, the `NEW` throws a `JNU_ThrowOutOfMemoryError` exception. However, the exception does not disrupt the control flow, and as a result the `pathv` variable in `splitPath` gets null. The subsequent "`pathv[count] = NULL`" will crash the JVM.

We classify this bug pattern as being security critical because dangerous operations in unexpected control-flow paths may enable an attacker to crash or take over the JVM.

## 3.2 C pointers as Java integers

Programs that use the JNI often need to pass C pointers through Java. Due to differences between Java's type system and C's, it is difficult (and sometimes impossible) for Java to assign types to C pointer values. The commonly used pattern in JNI programming is to cast C pointers to Java integers, and pass the resulting integers.

```
static void* xmalloc(JNIEnv *env, size_t size) {
  void *p = malloc(size);
  if (p == NULL) JNU_ThrowOutOfMemoryError(env, NULL);
  return p;
}

#define NEW(type, n) ((type *) xmalloc(env, (n) * sizeof(type)))

static const char * const * splitPath(JNIEnv *env, const char *path) {
  ...
  pathv = NEW(char*, count+1);
  pathv[count] = NULL;
  ...
}
```

Figure 3: An excerpt from solaris/native/java/lang/UNIXProcess_md.c. Even when `xmalloc` returns NULL, "pathv[count] = NULL" will be executed.

The pattern is used, for example, in the class `java.util.zip.Deflater`. The `Deflater` class supports compression using the Zlib C library. The Zlib library maintains a C structure (`z_stream`) for storing the state information of a compression data stream. A `Deflater` object holds a pointer to the `z_stream` structure, so that when the object calls Zlib the second time, the state information can be recovered through the pointer. As it is impossible for Java to declare the pointer as having the C type "`z_stream *`", the C code casts it into an integer before passing it to Java:

```
typedef struct z_stream_s {...} z_stream;

jlong Java_java_util_zip_Deflater_init
        ( ... ) {
  z_stream *strm =
    calloc(1, sizeof(z_stream));
  ... //initialize strm
  return (jlong) strm; //cast it to an integer
}
```

Whenever Java needs to access the compression stream, it passes to C the integer. C code then casts the integer back to a `z_stream` pointer, through which the state information of the stream can be retrieved or updated.

From Java's perspective, integers that represent C pointers are just ordinary Java integers. The pattern of treating C pointers as Java integers is unsafe if an attacker can inject to the C side arbitrary integer values that will be interpreted as pointers. Greenfieldboyce and Foster [17] examined the Gimp Toolkit (GTK) and discovered seven places where the injection of arbitrary integers is possible. For example, the native method `setFocus` in the GTK (shown below) has an integer parameter that represents a window pointer. Since the method is declared as a public method, an attacker can invoke it with an arbitrary integer value, which may corrupt memory and result in JVM crashes.

```
class GUILib {
  public native static void
    setFocus (int windowPtr);
  ...
}
```

We have built a custom scanner that searches for dangerous type casts from integers to pointers. The scanner is implemented in the CIL framework [30] as a CIL feature. We found 38 native methods that accept Java integers as arguments and then cast the integers to pointers. Compared to the GTK, the JDK's protection of these integers is safer. First, the native methods are all declared as private methods. An attacker cannot invoke them arbitrarily. Second, the Java integers that represent C pointers are stored in private fields.

If we assume Java's access control rules on private fields and methods are strictly enforced, then the JDK's protection on the integers is sufficient. However, with the Java reflection API, a Java program can at runtime change the private fields that store the C pointers, or invoke private methods.

If an attacker can use the Java reflection API, then he can read and write arbitrary memory locations by exploiting the pattern of C pointers as Java integers. For example, the `getAdler` native method (shown below) in the `java.util.zip.Deflater` class accepts a Java long, casts it to a pointer to the `z_stream` struct, and returns the `adler` field in the struct. If an attacker invokes it with the number that equals a target memory address minus the offset of the `adler` field, then he can read the value at the target address.

```
jint Java_java_util_zip_Deflater_getAdler
        (..., jlong strm) {
    return ((z_stream *)strm)->adler;
}
```

In a similar vein, the attacker can write to any memory location with his data through the `setDictionary` method in the `Deflater` class; the `setDictionary` method updates a `z_stream` structure with user-supplied data.

Although the default security policy when running untrusted Java code does not allow the Java reflection, we believe that passing C pointers as Java integers is dangerous, for the following reason. For a program in pure Java, an attacker can violate the access-control policy of the Java program (e.g. reading private fields) using the Java reflection, but the program remains type safe, which implies no reading/writing arbitrary memory locations. However, with the Java reflection *and* passing C pointers as Java integers through the JNI, an attacker could violate type safety by reading/writing arbitrary memory locations (shown by previous examples). We believe the privilege escalation from using the Java reflection to reading/writing arbitrary memory locations is a violation of the Java security model.

**Proposed fixes.** We recommend a fix based on an indirection table of pointers, similar to the OS file-descriptor table. The C side uses the indirection table to store pointers and passes table IDs, not pointers, to Java. When C gets the table IDs back from Java, it checks the validity of the IDs before carrying out dangerous operations. If bogus IDs were passed to C, the validity-checking step would catch it.

## 3.3 Race conditions in file accesses

Time-of-check-to-time-of-use (TOCTTOU) bugs refer to race conditions in which *"a program checks for a particular characteristic of an object, and then takes some action that assumes the characteristic still holds when in fact it does not"* [3]. Bishop and Dilger [3] identified a category of TOCTTOU bugs in file accesses. Such flaws occur, for example, when a program checks the access privilege of a file through a file path name and then use the file through the same file path name. Between the check and the use, if an attacker can change the file associated with the file path name, then the program may be fooled to access privileged files that the attacker cannot access otherwise.

We used ITS4 and Flawfinder to scan for file-access race conditions in the JDK. We identified three places in the target directories where file-access race conditions might occur. An example in `solaris/native/java/io/UnixFileSystem_md.c` is the race window between `stat` (line 144) and `chmod` (line 236). If the file in question were in a directory writable by the attacker, then during the race window he can link to that file any target file. The `chmod` at line 236 will then change the protection mode of the target file.

Besides the three race conditions we identified, we also discovered that the implementation of all the native methods in the class `java.io.UnixFileSystem` is based on path names, instead of file descriptors. For example, the `checkAccess` method checks whether the file or directory denoted by a given path name may be accessed; the `setPermission` method set on or off the access permission of the file or directory denoted by a given path name. The class `java.io.File`, a client of `java.io.UnixFileSystem`, uses `checkAccess` in methods such as `canRead` to check access permissions of a file path name stored in a field of `java.io.File`. It also uses `setPermission` in methods such as `setReadable` to set access permissions of the file path name. As a result, a Java program that uses `java.io.File` may have race conditions, if it first invokes `canRead`, and then invokes `setReadable`.

File-access race conditions are most relevant in a multi-user system, which is not a typical environment of using Java. Nevertheless, Java has been and will be used in a diverse set of scenarios (e.g., Java programs are run as root in the Java Authorization Toolkit [1]). Fixing the TOCTTOU flaws is usually straightforward. For example, the race window created by `stat` followed by `chmod` can be fixed by first opening the file to get its file descriptor, and then using `fstat` and `fchmod` on the file descriptor.

## 3.4 Buffer overflows

By automatically inserting array bounds checks, Java provides built-in protection against buffer overflows. If a program is developed in pure Java, we are rest assured that no buffer will be overflowed. However, since the implementation of the JDK contains C/C++ code, it is possible for an attacker to pass Java applications unexpected values, which flow to the C code in the JDK and trigger a buffer overflow.

Buffer overflows occur when a C program does not perform sufficient bounds checking. Native methods that use the JNI often need to check integers from Java for negative values. Since Java supports only signed integer types, the JNI maps all Java integer types to signed integer types in C. To use these signed integers safely for indices, sizes, and loop counters that should never have negative values, explicit checks are necessary. Missing checks for negative values may crash the JVM, as past bug reports have shown [19, 38].

We employed ITS4, Splint, and Flawfinder to scan the C files under the target directories for buffer-overflow bugs. ITS4 and Flawfinder scan for and report dangerous operations such as `strcpy`, `memcpy`, and `fscanf`. Splint reports many type-incompatibility warnings. For example, it issues a warning when a signed integer is used as an unsigned integer, which is helpful to identify missing checks for negative values. With the help of the static analysis tools, we discovered seven places where there are insufficient bounds checks. Two of them are in C functions that are not used by Java, and do not pose a security threat to the JVM. The rest pose real threats to the JVM: one bug is due to a missing width specifier in the format-string argument of `fscanf`; three bugs are due to possible integer overflows that may subsequently lead to buffer overflows; one bug is due to insufficient bounds checking of a public native method.[1]

## 3.5 Bugs related to dynamic memory management

The C code in the JDK needs to manage two memory regions, the C memory region and the Java memory region. It may mismanage both memory regions.

**Dynamic memory management in C.** Unlike Java, the C language provides programmers the power of manually managing memory through functions such as `malloc` and `free`. This power, which seems indispensable in system programming, has always been a constant source of programming defects, and consequently security vulnerabilities. Due to manual memory management, the C code in the JDK may suffer from a range of flaws, including dereferencing dangling pointers, multiple frees, and memory leaks. These defects may make the JVM unstable and vulnerable.

We employed Splint to identify defects related to memory management in the target directories. We manually inspected a large number of warnings and found only one case of memory leaks.

**Managing Java memory through the JNI.** Through the JNI, native methods can manage the Java memory. Certain JNI APIs manage Java memory in a style similar to `malloc` and `free`. For instance, to access a Java integer array, a native method first invokes `GetIntArrayElements` to have a pointer to the integer array. When the method finishes with the array, it is supposed to invoke `ReleaseIntArrayElements` to release the pointer. These JNI API functions enable the C method to communicate with Java's Garbage Collector (GC). `GetIntArrayElements` informs the GC of the creation of a C pointer to the Java array; the GC should not garbage collect or move the array. `ReleaseIntArrayElements` informs the GC that the C pointer is no longer needed.

This style of manual memory management is error prone and has similar problems to the ones of `malloc/free`. For example, using the C pointer after `ReleaseIntArrayElements` is similar to using a dangling pointer, since the Java GC may have already moved or garbage collected the array. Failure to invoke `ReleaseIntArrayElements` will make the GC retain the array indefinitely. Other pairs of functions that are similar to `Get/ReleaseIntArrayElements` include `Get/ReleaseStringUTFChars`, `New/DeleteGlobalRef`, and `Push/PopLocalFrame`.

We developed grep-based scripts to pattern match places where relevant JNI API functions such as `ReleaseIntArrayElements` are used. In the target directories, we discovered one place where `ReleaseStringUTFChars` is not invoked (in one control-flow path) to release a Java String reference. There are also 27 places where JNI global references are not released.[2] Although these bugs are not security critical, they result in memory leaks and are worth fixing.

## 3.6 Insufficient error checking

One of the most common mistakes when writing C code is missing checks for error cases. Since the C language does not have an exception mechanism, programmers are required to perform explicit checks after many function calls that may return special values for reporting errors. For instance, the standard `malloc` function returns a null value if the required space cannot be allocated. The correct usage of the `malloc` function should first check the return value for nonnull before using it. We encountered two places where the C code in the target directories forgets the check for the `malloc` function.

In addition, many JNI API functions use null values to report errors. For example, the `GetFieldID` function returns null when the operation fails.[3] The following code crashes Sun's JVM, when `fid` gets null.

```
//Get the field ID
fid=(*env)->
      GetFieldID(env, cls, "x", "I");
//Get the int field
int i=(*env)->GetIntField(env, obj, fid);
```

The above code should first check `fid` to be nonnull, before invoking `GetIntField`.

We developed grep-based scripts to scan for JNI API functions whose return values should be checked. We inspected suspicious JNI API calls to check whether their return values are checked before used. In total, we found

| JNI API FUNCTIONS | # OF VIOLATIONS |
|---|---|
| GetFieldID/GetStaticFieldID | 5 |
| GetMethodID/GetStaticMethodID | 3 |
| GetStringUTFChars | 4 |
| FindClass | 11 |
| New⟨Type⟩Array | 1 |
| NewGlobalRef | 11 |
| Total | 35 |

Table 2: Insufficient error checking. For each JNI API, the table lists the number of cases in the target directories where there is no checking of the return value of the API before using the value.

35 violations. Table 2 summarizes the results in the target directories. Note that the table does not include those JNI API functions for which we did not find violations. We consider insufficient error checking to be security critical because they may result in JVM crashes.

## 3.7 Other flaws resulting from misusing the JNI

For completeness, we next mention other bug patterns in the native code of the JDK. For these patterns, we either have not found any bugs in the target directories, or have not successfully applied static analysis tools.

**Type misuses.** The JNI maps Java types to C/C++ types and performs necessary conversions when data go through the interface. Java primitive types and Java reference types are mapped differently. Java primitive types are mapped directly. For example, the Java type `int` is mapped to the native type `jint` (declared as 32-bit integers in jni.h). Java objects of reference types are mapped to *opaque references*, which are pointers to internal data structures in the JVM. The exact layout of the internal data structures is hidden from programmers. In C, all opaque references have the type `jobject`. Native C/C++ code manipulates these references through JNI API functions.

Since native C code treats all references to Java objects as having one single type[4], C compilers cannot distinguish references to objects of different Java classes. As a result, an object of Java class `A` may be wrongly passed to a JNI API function that actually requires an object of Java class `B`. Type checking in C compilers cannot catch this kind of mistakes, which usually results in JVM crashes. More serious is the case that a native method invokes a Java method with objects of wrong classes. A type confusion like this could have serious consequences, as past research on Java security has shown [32, 6].

Another case of type misuses in the JNI is that programmers may invoke wrong JNI API functions. For example, programmers may use wrong JNI array APIs, as the JNI provides different APIs for accessing arrays of different types. There are `GetByteArrayElements`, `GetIntArrayElements`, and others. Calling wrong JNI API functions may result in improper memory accesses or JVM crashes.

JSaffire [15] by Furr and Foster is a tool that can check type misuses in the JNI code. We did not incorporate JSaffire into our step of characterizing bug patterns for two reasons. First, this category of bugs has been well characterized in previous work [15, 34]. Second, we suspect type-misuse bugs in the JDK's native code would be rare. Type-misuse bugs usually result in immediate program crashes and are easy to trigger with a small amount of test code. As the JDK has been extensively "tested" by its users, we believe that most of the type-misuse bugs have been fixed. This is partly confirmed by our experiment. We constructed scripts to search for the most common cases of type-misuse bugs, such as passing wrong classes to JNI API functions and confusing jclass with jobject [24, ch 10.3]; we did not find any such kinds of bugs in the target directories.

**Deadlocks.** The JNI includes pairs of functions `Get/ReleaseStringCritical` and `Get/ReleasePrimitiveArrayCritical`, which introduce a critical region. Inside the region, the C code cannot issue blocking calls or allocate new Java objects. Otherwise, the JVM may deadlock. We inspected all such critical regions in the target directories and did not find any risk of deadlock.

**Violating the Java security model.** The JNI does not enforce access controls on classes, fields, and methods that are expressed in the Java language through the use of modifiers such as `private`. Therefore, a native method can read private fields of any Java object. Furthermore, a native method can violate the Java sandbox security model, by performing dangerous operations that would otherwise be blocked by the JVM. We have not checked the JDK's native code for these kinds of violations.

## 4 Remedies, limitations, and directions

The native code inside the JDK is critical to Java security. As we and others have demonstrated, after more than a decade, there are still flaws remaining in this critical code. Once identified, these flaws are generally not hard to fix. However, the perpetual mode of patching is less than satisfying. Next we discuss more systematic approaches, their limitations, and future directions.

## 4.1 Static analysis

Static analysis is useful for isolating and eliminating security bugs, as demonstrated by the number of bugs we identified with the help of static analysis tools. On the other hand, there are a few limitations of the current generation of static analysis tools that prevent us from using them to cover all 800k lines of native code in the JDK.

**Limitations of static analysis tools.** The tools we used issued a large number of warnings that are false positives. For each of the three off-the-shelf tools, the following table lists the number of warnings it issued, the number of true errors, and its false-positive rate.

| Off-the-Shelf Tools | Warnings | Errors | FP rates |
|---|---|---|---|
| ITS4 -c1 | 241 | 6 | 97.5% |
| Flawfinder | 297 | 5 | 98.3% |
| Splint[5] | 3532 | 7 | 99.8% |

Our own scripts and scanners perform slightly better, but the false-positive rates are still high; see Table 3.

Due to the large number of false positives, we had to manually sift through many cases—the principal reason why we examined only a portion of the native code in the JDK. In addition to false positives, static analysis tools may have false negatives. For example, of the four buffer-overflow bugs identified in the target directories, ITS4 and Flawfinder missed one and Splint missed two.

Another limitation of the static analysis tools is that they analyze C code alone, without considering how the Java side interacts with the C side. This is a severe limitation because the interface code between Java classes and C libraries is where most bugs arise. In fact, all the bugs we identified are in the interface code. This is not only because the two libraries in the target directories (namely, Zlib and fdlibm) have been used in many other applications besides the JDK and are mature, but because programmers tend to make wrong assumptions of the Java and C sides when writing interface code.

When analyzing the interface code, considering both sides of Java and C can significantly increase analysis precision and reduce false positives and negatives. To illustrate, we use the `java.util.zip.Deflater` class as an example. The public `deflate` method shown in Figure 4 accepts a buffer, an offset, and a length from users, and then invokes the native method `deflateBytes`. To be safe, the `deflate` method checks the bounds of the offset and the length parameters before invoking the native method `deflateBytes`.

For the example in Figure 4, a static analysis that analyzes only C code has to make either an optimistic or a pessimistic assumption about whether the Java side has performed the bounds checking. If the analysis makes the optimistic assumption, it would produce false negatives if the Java side had forgotten to check the bounds. If it makes the pessimistic assumption, it would have to flag any access to the `b` buffer through the offset and the length as a possible error. For example, the `SetByteArrayRegion` operation in `deflateBytes` would be flagged as a possible out-of-bounds array write, even though that is impossible given the Java context. Bug finders usually make pessimistic assumptions for the purpose of minimizing false negatives. For instance, Splint flags "`malloc(len)`" in `deflateBytes` and complains about an incompatible type cast from the signed integer `len` to an unsigned integer expected by `malloc`—it does not know that the Java side invokes `deflateBytes` only with positive lengths.

The necessity of inter-language analysis is also sharply enforced by our experience of manual inspection. For many warnings, we inspected both their C and Java contexts to decide if they are true errors. To give a rough idea of how many warnings cannot be eliminated as false positives without taking the Java context into account, we examined the 139 incompatible-type-cast warnings that Splint issued for the C code under `java.util.zip` and found that in 22 cases the Java context must be inspected.

**Future directions of improving static analysis tools.** Some of the limitations we mentioned are particular to the tools we used, and are not fundamental to static analysis. The off-the-shelf tools used in this study are known for having high false-positive rates. ITS4, Flawfinder, and our own tools are based on simple syntactic pattern matching; Splint performs certain type-based analyses, but is still a coarse-grained tool. We believe false-positives rates can be significantly reduced through advanced static techniques such as software model checking (e.g., MOPS [4], CMC [29], SLAM [2]), and Bandera [7]), type qualifiers [12, 17], theorem proving techniques (e.g., ESC/Java [11]), and others.

To better analyze interface code, we advocate *inter-language analysis* across Java and C. Most existing tools are limited *a priori* to code written in a single language. Few inter-language analyses across Java and C exist. JSaffire [15] is an exception, but can only check for type misuses of data from Java to C. Our previous work, ILEA [36], enables general inter-language analysis across Java and C. The basic approach of ILEA is to perform a partial compilation from C code to a specification based on Java so that an existing Java analysis can understand the behavior of the C code through the Java specification. ILEA extends Java with a set of simple, yet powerful approximation primitives, which enable auto-

| Bug patterns | Our tools | Warnings | Errors | FP rates |
|---|---|---|---|---|
| Unexpected control flows due to mishandling exceptions | grep-based scripts | 337 | 11 | 96.7% |
| C pointers as Java integers | scanner built in CIL | 46 | 38 | 17.4% |
| Mem. management flaws (Java Mem.) | grep-based scripts | 43 | 28 | 34.9% |
| Insufficient error checking (JNI APIs) | grep-based scripts | 230 | 35 | 84.8% |

Table 3: False-positive rates of our tools.

java.util.zip.Deflater

```
public class Deflater {
  public synchronized int deflate(byte[] b, int off, int len) {
    ...
     if (off < 0 || len < 0 || off > b.length - len) {
       throw new ArrayIndexOutOfBoundsException();
     }
    return deflateBytes(b, off, len);
  }

  private native int deflateBytes(byte[] b, int off, int len);
}
```

C implementation of deflateBytes()

```
jint Java_java_util_zip_Deflater_deflateBytes
       (JNIEnv *env, jobject this, jarray b, jint off, jint len) {
  ...
  out_buf = (jbyte *) malloc(len);
  ...
  (*env)->SetByteArrayRegion(env, b, off, len - strm->avail_out, out_buf);
  ...
}
```

Figure 4: An example illustrating the necessity of inter-language analysis

matic extraction of partial Java specifications of C code. Through ILEA, any existing analysis on Java in principle can be extended to also cover C code. In practice, however, ILEA is restricted by its compilation precision, and also by the effectiveness of the Java analysis.

We plan to combine advanced static analysis techniques with the ideas in ILEA to build high-precision, inter-language tools that hunt for bugs in the JDK's native code. We are particularly interested in taint analysis and software model checking. Static taint analysis (e.g., [26]) can track attacker-controllable data that flow from Java to C. Software model checking can check for violations of many patterns we have discussed as they can be formalized as state machines. We plan to investigate C model checkers such as MOPS [4] and CMC [29] and extend them to perform inter-language checking using the ideas in ILEA.

Finally, we believe it is important to formalize the soundness proofs of static analysis tools. Formal study helps understand the assumptions, clarify guarantees, and reduce false negatives. In the context of the JNI, formal study is complicated by the lack of formal semantics of the C language. It is perhaps helpful to focus instead on a well-defined subset of C such as Cminor [23].

## 4.2 Dynamic Mechanisms

Static analysis analyzes programs to find implementation errors before the programs are run. An alternative is to use dynamic mechanisms to prevent or isolate errors during runtime. Dynamic mechanisms can take advantage of richer runtime information to check certain properties easily, although sacrificing some performance.

Our previous work, SafeJNI [34], is a mostly dynamic mechanism for ensuring the safety of JNI-based programs such as the JDK. It first leverages CCured [31] to provide internal memory safety to the C code. CCured analyzes C programs to identify places where memory safety might be violated and then inserts runtime checks to ensure safety. SafeJNI also inserts runtime checks at the boundary between Java and C to make sure that the C code accesses the Java state safely and cooperates with Java's garbage collector. SafeJNI incurs a performance overhead of 14–119% on a set of microbenchmark programs, and incurs 63% on Zlib.

Table 4 summarizes how SafeJNI protects Java from bugs in the native code in terms of the various bug patterns discussed before. SafeJNI protects Java from most kinds of bugs in the native code. Its main limitation is that it does not protect against concurrency-related bugs (race conditions and deadlocks); we believe concurrency-related bugs should be best addressed through advanced static analysis techniques.

**Future directions.** We believe that SafeJNI is a promising direction to prevent errors in the native code. We plan to reduce its overhead in two ways. First, static analysis techniques can reduce a large number of dynamic checks. For example, many runtime type checking can be eliminated if we can statically track the classes of Java objects in C, similar to what JSaffire does [14].

Second, we plan to explore other more efficient ways of providing internal safety to C code than CCured. Our experiment showed that CCured accounted for most of the performance overhead in SafeJNI (46% out of 63% in Zlib). The relatively large performance slowdown is because CCured guarantees every C buffer is well protected. For instance, given the code below

```
int *p = (int *) malloc (1024);
*(p+i) = 3;
```

CCured in general will insert the runtime check "0 <= i < 1024" before "*(p+i) = 3".

If the safety policy is to protect the JVM state from being accidentally destroyed by C code, then Software Fault Isolation (SFI [40, 27]) of the C code is sufficient. Whenever the JVM starts to execute a native method, it can first allocate a trunk of memory, say 16MB, and hand the memory region to the native method. A SFI-based scheme can then guarantee that any access of the C memory will not escape the memory region, and thus will not destroy the JVM state.

Schemes based on SFI can isolate errors within native components, but does not prevent exploits of vulnerabilities inside the components. XFI [9], on the other hand, can prevent exploits of a large number of vulnerabilities by enforcing properties such as control-flow integrity. In addition, it works on assembly code and is not restricted to a source programming language.

## 4.3 Reimplementation in safer languages

It can be argued that the C language is intrinsically unsafe and should not be used in the JDK. In the long run, we believe the C code in the JDK should be reimplemented in safer languages. The obvious choice is Java. This is a feasible approach, as there exist implementations in pure Java of many programs originally written in C, such as the Zlib library [20]. GNU Classpath, an open-source replacement of Sun's JDK, takes this approach seriously; one of their long-term goals is to become JNI independent by implementing everything in Java [5]. On the flip side, rewriting the existing 800 kloc of C/C++ code in Java will require a substantial investment, and will likely have a negative impact on execution speed.

Another idea is to use a safe C variant to port the C code. Cyclone [21] is a reasonable choice. Since the syntax and semantics of Cyclone are close to C, porting C code to Cyclone should take less time than, say, a complete rewrite in Java. However, as Cyclone has a strong type system and uses region-based memory management, converting to type-checkable Cyclone code will not be a trivial effort. Furthermore, this approach alone can guarantee only the internal safety of C code. The C code can still misuse the JNI interface.

Since the JNI interface is extraordinarily verbose and error prone, one approach to reducing flaws is to use a better interface between Java and C. A notable example is Jeannie [18], which allows programmers to write mixed Java and C code in a single file. The Jeannie compiler then translates mixed Java/C code into code that uses the JNI. Although in Jeannie it is still possible to write unsafe code, Jeannie helps programmers reduce errors. For example, in Jeannie programmers can raise Java exceptions directly, thus avoiding the control-flow problem when raising JNI exceptions (Section 3.1).

## 5 Conclusion

The large amount of native code in the JDK is a time bomb in Java security. Our study has examined a range of bug patterns in the JDK's native code, from well-known buffer overflows to new patterns such as unexpected control flow paths due to mishandling JNI exceptions. Given the importance of Java, it is imperative to develop better, inter-language static and dynamic mechanisms to mediate the threats posed by the native code.

Through our study, we hope to send the message that the native code should be kept at a minimum in the JDK. On the contrary, the native code in Sun's JDK has been on the increase. The native code is outside of the Java security model and defeats Java's main goals: safety, security, and platform independence. In the long run, most

| Bug patterns | | How SafeJNI works against the bugs? |
|---|---|---|
| Unexpected control flows due to mishandling exceptions | | Through SafeJNI's dynamic checks on pending exceptions. |
| Race conditions in file accesses | | N/A |
| Buffer overflows | | Through CCured and SafeJNI's static pointer kind system. |
| Mem. management flaws | C mem. | Through CCured. |
| | Java mem. | Through SafeJNI's memory management scheme. |
| Insufficient error checking | JNI APIs | Through SafeJNI's dynamic checks. |
| | misc. | Through CCured. |
| Type misuses | | Through SafeJNI's dynamic checks. |
| Deadlocks | | N/A |
| Violating the Java security model | | Partly addressed through SafeJNI's dynamic checks on access-control rules on Java fields/methods. |

Table 4: How SafeJNI protects the JVM from bugs?

of the native code should be ported to safer languages such as Java.

## Acknowledgments

We would like to thank Andrew Appel and Edward Felten for their comments; and to anonymous reviewers for their constructive feedbacks on earlier versions of this paper; and to Xiaolan Zhang and Angelos Stavrou for shepherding the paper to its final form.

## References

[1] Overview of authorization toolkit for Java. Retrieved Apr 26th, 2008, from `http://www.amug.org/~glguerin/sw/authkit/overview.html`.

[2] Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 203–213.

[3] Bishop, M., and Dilger, M. Checking for race conditions in file accesses. *Computing Systems 9*, 2 (1996), 131–152.

[4] Chen, H., and Wagner, D. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)* (2002), pp. 235–244.

[5] Classpath decisions. Retrieved Apr 26th, 2008, from `http://developer.classpath.org/mediation/ClasspathDecisionsPage`.

[6] Coglio, A., and Goldberg, A. Type safety in the JVM: some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience 13*, 13 (2001), 1153–1171.

[7] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)* (2000), pp. 439–448.

[8] Dean, D., Felten, E. W., and Wallach, D. S. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy (S&P)* (1996), pp. 190–200.

[9] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *OSDI* (2006), pp. 75–88.

[10] Evans, D., and Larochelle, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (2002), 42–51.

[11] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 234–245.

[12] Foster, J. S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2002), ACM Press, pp. 1–12.

[13] Freund, S. N., and Mitchell, J. C. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning 30*, 3-4 (2003), 271–321.

[14] Furr, M., and Foster, J. S. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2005), pp. 62–72.

[15] Furr, M., and Foster, J. S. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)* (2006), pp. 309–324.

[16] GNU GLOBAL source code tag system. `http://www.gnu.org/software/global/`.

[17] Greenfieldboyce, D., and Foster, J. S. Type qualifier inference for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007). 321-336.

[18] Hirzel, M., and Grimm, R. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007), pp. 19–38.

[19] Java bug report 4153825. `http://bugs.sun.com/view_bug.do?bug_id=4153825`.

[20] JCraft. `http://www.jcraft.com/`.

[21] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (2002), USENIX Association, pp. 275–288.

[22] Klein, G., and Nipkow, T. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. on Programming Languages and Systems 28*, 4 (2006), 619–695.

[23] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages (POPL)* (2006), pp. 42–54.

[24] Liang, S. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] Liang, S., and Bracha, G. Dynamics class loading in the Java virtual machine. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 1998), ACM, pp. 36–44.

[26] Livshits, V. B., and Lam, M. S. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium* (Aug. 2005), pp. 271–286.

[27] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (2006).

[28] McGraw, G., and Felten, E. W. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[29] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A pragmatic approach to model checking real code. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[30] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)* (2002), pp. 213–228.

[31] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)* (2002), pp. 128–139.

[32] Saraswat, V. Java is not type safe, 1997.

[33] Schoenefeld, M. Denial-of-service holes in JDK 1.3.1 and 1.4.1_01. Retrieved Apr 26th, 2008, from `http://www.illegalaccess.org/java/ZipBugs.php`, 2003.

[34] Tan, G., Appel, A. W., Chakradhar, S., Raghunathan, A., Ravi, S., and Wang, D. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering* (2006), pp. 97–106.

[35] Tan, G., and Croft, J. An empirical security study of the native code in the JDK. Tech. rep., Boston College, May 2008. Available on the first author's website.

[36] Tan, G., and Morrisett, G. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007), pp. 39–56.

[37] US-CERT. Vulnerability note VU#138545: Java Runtime Environment image parsing code buffer overflow vulnerability, June 2007. Credit goes to Chris Evans.

[38] US-CERT. Vulnerability note VU#939609: Sun Java JRE vulnerable to arbitrary code execution via an unspecified error, Jan. 2007. Credit goes to Chris Evans.

[39] Viega, J., Bloch, J. T., Kohno, Y., and McGraw, G. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference (ACSAC)* (2000).

[40] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles* (New York, 1993), ACM Press, pp. 203–216.

[41] Wallach, D. S., and Felten, E. W. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy (S&P)* (1998), pp. 52–63.

[42] Wheeler, D. A. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

## Notes

[1]This bug is not in the target directory and was found in a casual inspection.

[2]Global references are never released in the code we examined, although the JNI manual explicitly mentioned the necessity of freeing global references [24, ch5.2.3].

[3]It fails if the specified field cannot be found, or if the class initializer fails, or if the system runs out of memory [24].

[4]In C++, certain Java built-in classes have corresponding C++ classes in the JNI (predefined in jni.h). References to objects of other Java classes, including all user-defined classes, are still mapped to `jobject`.

[5]With the options "+posixlib -paramuse -redef -noeffect -varuse -exportlocal -incondefs -booltype jboolean -booltrue JNI_TRUE -boolfalse JNI_FALSE -predboolint -compdef".

# AutoISES: Automatically Inferring Security Specifications and Detecting Violations

Lin Tan[1]
*University of Illinois, Urbana-Champaign*
*lintan2@cs.uiuc.edu*

Xiaolan Zhang
*IBM T.J. Watson Research Center*
*cxzhang@us.ibm.com*

Xiao Ma[†‡], Weiwei Xiong[†], Yuanyuan Zhou[†‡]
[†]*University of Illinois, Urbana-Champaign* [‡]*Pattern Insight Inc.*
{*xiaoma2, wxiong2, yyzhou*}*@cs.uiuc.edu*

## Abstract

The importance of software security cannot be overstated. In the past, researchers have applied program analysis techniques to automatically detect security vulnerabilities and verify security properties. However, such techniques have limited success in reality because they require manually provided code-level security specifications. Manually writing and generating these code-level security specifications are tedious and error-prone. Additionally, they seldom exist in production software.

In this paper, we propose a novel method and tool, called A*uto*ISES, which Automatically Infers Security Specifications by statically analyzing source code, and then directly use these specifications to automatically detect security violations. Our experiments with the Linux kernel and Xen demonstrated the effectiveness of this approach – AutoISES automatically generated 84 security specifications and detected 8 vulnerabilities in the Linux kernel and Xen, 7 of which have already been confirmed by the corresponding developers.

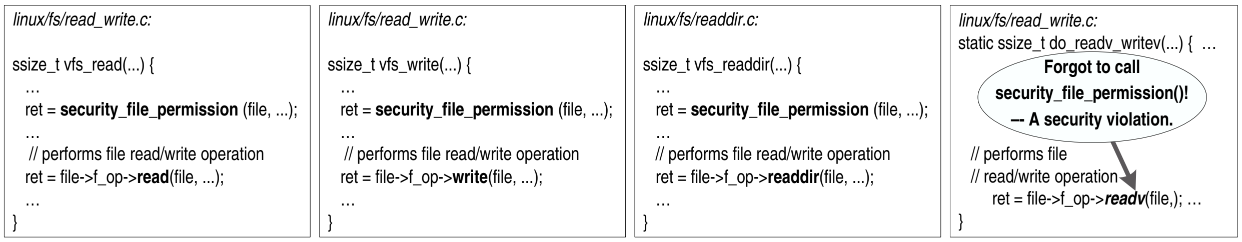
## 1 Introduction

### 1.1 Motivation

The critical importance of software security has driven the design and implementation of secure software systems. Security-Enhanced Linux (SELinux) [23, 28], developed as a research prototype to incorporate Mandatory Access Control (MAC) into the Linux kernel several years ago, imposes constraints on its existing Discretionary Access Control (DAC) for stronger security. SELinux has since been adopted by the mainline Linux 2.6 series and incorporated into many commercial distributions, including Redhat, Fedora, and Ubuntu. Recently, Xen also adopted a similar MAC security architecture to enable system-wide security policy [7].

A core part of such access control systems is a set of security check functions, which check whether a subject (e.g., a process) can perform a certain operation (e.g., read or write) on an object (e.g., a file, an inode, or a socket). These protected operations are called *security sensitive operations*. For example, Linux's security check function `security_file_permission(`$\langle file \rangle$`, ...)` can determine if the current process is authorized to read or write the file, while another security check function, `security_file_mmap(`$\langle file \rangle$`, ...)`, checks if the current process is authorized to map a file into memory. To ensure only authorized users can read or write the file, developers must add the security check function `security_file_permission()` before each file read/write operation on every file. Similarly, developers must add `security_file_mmap()` each time before mapping a file to memory, to ensure only authorized users can memory map the file.

A major challenge of supporting the secure architecture above is to ensure that all sensitive operations on all objects are protected (i.e., checked for authorization) by the proper security check functions in a consistent manner. If the proper security check function is missing before a sensitive operation, an attacker with insufficient privilege will be able to perform the security sensitive operation, causing damage. For example, the file read/write operation is performed in many functions throughout the Linux kernel, such as `read()`, `write()`, `readv()`, `writev()`, `readdir()`, and `sendfile()`. Despite the different names of these function calls, they all perform the same *conceptual* file read/write operation, and must be checked for authorization by calling the security check function `security_file_permission()`. As the Linux kernel code is reasonably mature, most of these functions performing the file read/write operation, such as `read()`, `write()`, and `readdir()`, are protected by



```
linux/fs/read_write.c:

ssize_t vfs_read(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
   // performs file read/write operation
  ret = file->f_op->read(file, ...);
  ...
}
```

(a) file read/write operation protected by the check

```
linux/fs/read_write.c:

ssize_t vfs_write(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
   // performs file read/write operation
  ret = file->f_op->write(file, ...);
  ...
}
```

(b) file read/write operation protected by the check

```
linux/fs/readdir.c:

ssize_t vfs_readdir(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
  // performs file read/write operation
  ret = file->f_op->readdir(file, ...);
  ...
}
```

(c) file read/write operation protected by the check

```
linux/fs/read_write.c:
static ssize_t do_readv_writev(...) {  ...

  // performs file
  // read/write operation
       ret = file->f_op->readv(file,); ...
}
```

(d) **Security Violation –** violating the implicit security specification

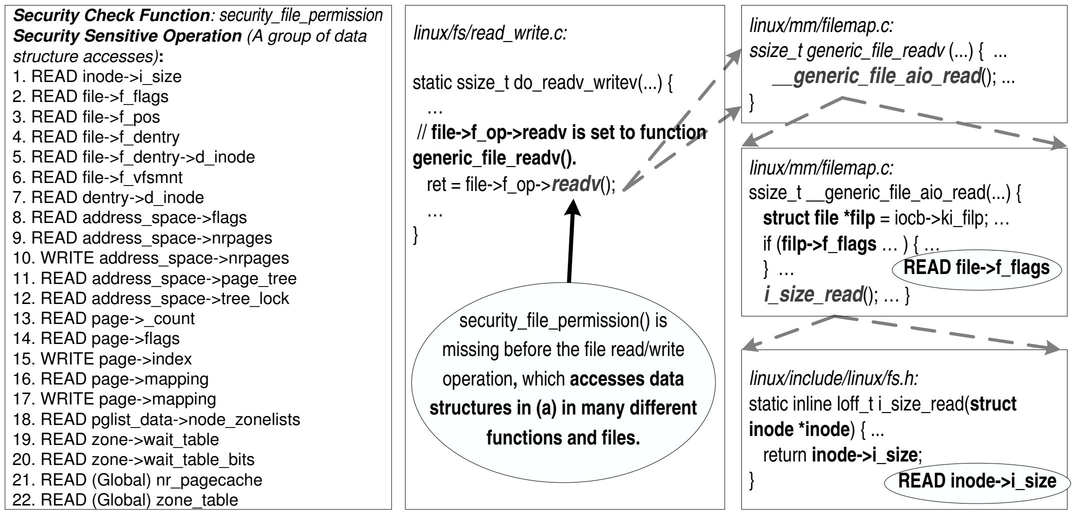
Figure 1: A real security violation in Linux 2.6.11. The security check `security_file_permission()` was missing before the security sensitive operation performed via `file->f_op->readv()`, violating the implicit security specification — every file read/write operation must be checked for authorization using `security_file_permission()`. This is a real security violation, which has already been fixed in later versions. The code is slightly modified to simplify illustration.

the security check function, as shown in Figure 1(a)-(c). However, in a few other cases, as shown in Figure 1(d), the security check function is not invoked before the file read/write operation performed by `readv()`, violating the *implicit* security specification or security rule: every file read/write operation must be protected by calling security check function `security_file_permission()`. Due to this real world security vulnerability in Linux 2.6.11 (*CVE-2006-1856* [1]), unauthorized user can read and write files that they are not allowed to access, potentially providing unauthorized user account access. Additional damages might include partial confidentiality, integrity, and availability violation, unauthorized disclosure of information, and disruption of service.

There have been great advances in applying program analysis techniques [2, 4, 5, 12, 16] to automatically detect these security vulnerabilities and to verify security properties [6, 9, 18, 30]. Generally, these tools take a specification that describes the security properties to verify as input. For example, in earlier efforts [9, 30], the authors *manually* identified the data types (e.g., `struct file, struct inode`, etc.) that might be accessed to perform security sensitive operations and automatically verified that any access to these data types was protected by a security check function. Although these previous studies detected some vulnerabilities, and made significant progresses toward automatic verification of security properties, they are limited in two perspectives:

- All these previous tools require developers or their tool users to provide code-level security specifications, which greatly limit their practicability in checking and verifying security properties. Writing specifications that accurately capture the security properties of a piece of software and at the same time maintaining their correctness across different versions of the software is notoriously difficult. Such specifications seldom exist in production software.
- Human-generated specifications can be imprecise, causing false positives and potentially false negatives in violation detection. As an example, the specification used in one of the earlier work [30], introduced false positives because it treated *any* access to specified data structures as security sensitive operations. In reality, a security sensitive operation typically consists of accesses to multiple data structures: a file read/write operation involves accessing `struct file`, `struct inode`, `struct dentry`, etc. Accessing the file structure alone is not necessarily (actually in most cases is not) a file read/write operation. In addition, some field accesses (e.g., `file->f_version`) of a security sensitive data type are not part of any security sensitive operation, and therefore do not need to be protected. Therefore, in the two cases above, simply requiring accesses to every field of these data structures to be protected led to false positives [30]. The specification may also introduce false negatives because it does not specify which security check is required for which operation. The tool can fail to detect violations where the wrong check function is used as different security sensitive operations (e.g., file read/write and memory map) may access the same data types (e.g., `struct file`) but require different security checks.

Therefore, to design tools that are truly usable for ordinary programmers, it is highly desirable for these tools to meet the following three requirements on specification generation: (1) to automatically check against source code for security violations, the security specifications must be at the code level. The conceptual specification "file read/write operation must be protected by the security check function `security_file_permission()`" can not be checked against source code without knowing its corresponding code-level representation. (2) as it is tedious and error-prone for developers to write these



**Security Check Function**: security_file_permission
**Security Sensitive Operation** (A group of data structure accesses):

```
1. READ inode->i_size
2. READ file->f_flags
3. READ file->f_pos
4. READ file->f_dentry
5. READ file->f_dentry->d_inode
6. READ file->f_vfsmnt
7. READ dentry->d_inode
8. READ address_space->flags
9. READ address_space->nrpages
10. WRITE address_space->nrpages
11. READ address_space->page_tree
12. READ address_space->tree_lock
13. READ page->_count
14. READ page->flags
15. WRITE page->index
16. READ page->mapping
17. WRITE page->mapping
18. READ pglist_data->node_zonelists
19. READ zone->wait_table
20. READ zone->wait_table_bits
21. READ (Global) nr_pagecache
22. READ (Global) zone_table
```

```
linux/fs/read_write.c:

static ssize_t do_readv_writev(...) {
  ...
 // file->f_op->readv is set to function
generic_file_readv().
  ret = file->f_op->readv();
  ...
}
```

security_file_permission() is missing before the file read/write operation, which accesses data structures in (a) in many different functions and files.

```
linux/mm/filemap.c:
ssize_t generic_file_readv (...) {  ...
     __generic_file_aio_read(); ...
}
```

```
linux/mm/filemap.c:
ssize_t __generic_file_aio_read(...) {
   struct file *filp = iocb->ki_filp; ...
   if (filp->f_flags ... ) { ...
   }  ...
   i_size_read(); ... }
```

```
linux/include/linux/fs.h:
static inline loff_t i_size_read(struct
inode *inode) { ...
   return inode->i_size;
}
```

(a) Security Rule/Specification (b) Security Violation (c) Accesses in the Code

Figure 2: A code-level security specification AutoISES automatically generated and a real security violation to the specification in Linux 2.6.11. The leftmost box shows the security rule, consisting of a security check function and a group of data structure accesses. Each row is one access, which can be either a structure field access or a global variable access, denoted by "(Global)". For each structure field access, the name before the first −> is the type name of the structure, and the rest are field names. For a global variable, the variable name is used. The code is slightly modified to simplify illustration.

security rules, the tool should automatically generate security specifications with minimum user/developer involvement; and (3) the generated specification must be precise, otherwise it would result in too many false positives and/or false negatives.

## 1.2 Our Contributions

This paper makes two contributions:

**(1) We propose an approach and a tool, AutoISES, to automatically extract concrete code-level security specifications from source code and automatically detect security violations to these specifications.** Our key observation is that although the same security sensitive operation can be performed in different functions, ultimately, the structure fields and global variables these functions access are the same. We call these structure field and global variable accesses together as *data structure accesses*. For example, all of the different functions performing the file read/write operation share the 22 data structure accesses listed in Figure 2(a) (automatically generated by AutoISES), including reading field `f_flags` of the `file` structure and reading field `i_size` of the `inode` structure. These 22 data structure accesses are performed in many different functions located in different source files. Intuitively, this makes sense as security check functions are designed to protect *data*. Therefore, the use of data structure accesses is *fundamental* in representing security sensitive operations.

Based on this observation, we propose a method and a tool, called A*uto*ISES, to Automatically Infer Security Specifications by statically analyzing source code and *directly* use these specifications to automatically detect security violations. Specifically, if a code-level security sensitive operation is frequently protected by a security check function in source code, AutoISES automatically infers that the security check function should be used to protect the particular code-level security sensitive operation. Our *rationale* is that for release software, the majority of the code should be correct, therefore we can use the code to infer security specifications or rules, which are observed in most places of the source code, but may not in a few other places. The rationale is similar to that of prior work in specification mining [10, 11, 20, 22], each of which extracts different types of programming rules automatically from source code or execution trace. However, *previous techniques are not directly applicable to our problem*, because they are limited by the types of rules they can infer (e.g., function correlation rules [10, 20], variable value related invariants [11], variable pairing rules [22]). As noted previously, our key observation states that the sensitive operation should be represented as data structure accesses, therefore, it requires AutoISES to be able to learn specifications that contain both functions and multiple variable accesses that satisfy certain constraints. None of the previous techniques can be applied without significant re-design of the learning algorithm (more detailed discussion in Section 6.1 and Section 7).

We evaluated AutoISES on the latest versions of two large software systems, the Linux kernel and Xen, to demonstrate the effectiveness of our approach. AutoISES automatically extracted 84 rules from the Linux kernel and Xen, and detected 8 true violations, 7 of which are confirmed and fixed by the corresponding developers. Figure 2 shows (a) the code-level security specification learned by AutoISES which consists of the 22 data structure accesses, (b) a security violation automatically detected by AutoISES, and (c) the unprotected sensitive operation that performs all the accesses shown in (a) in different functions located in various source files. It would be very difficult, if impossible, for a human being to generate such a specification. More examples and results can be found in Section 5.

The automatically generated specifications can also be used by other analysis tools for vulnerability detection. Additionally they can assist in software understanding and maintenance. These results demonstrate that AutoISES is effective at automatically inferring security rules and detecting violations to these rules, which greatly improves the practicality of security property checking and verification tools.

**(2) We quantitatively evaluate rule granularity impact on the accuracy of rule inference and violation detection.** Security specifications can vary in *granularity*. For example, a single access can be represented with the access type (read or write), `READ inode->i_size`, or without it, `ACCESS inode->i_size`. Similarly, the same access can be represented with the structure field, `READ inode->i_size`, or without it, `READ inode`. Theoretically, finer granularity causes more false negatives and fewer false positives for violation detection compared to coarser granularity. The choice of granularity can greatly affect the accuracy of rule inference and violation detection. Almost all previous rule generation and violation detection techniques [3, 9, 10, 11, 20, 22, 30] choose fixed granularity without *quantitatively* evaluating how good their choice is.

In our work, we quantitatively evaluate the impact of different rule granularity on rule inference and violation detection. This approach is *orthogonal* to our automatic rule inference techniques and can be applied to other rule extraction techniques.

Interestingly, our results show that if we do not distinguish the fields, then the inferred code-level security sensitive operation for the check function `security_inode_link()` and `security_file_unlink()` is the same, failing to distinguish the two different operations. Using our finest granularity, AutoISES can disambiguate the two similar operations (the unlink operation contains `READ inode->i_size`, but the link operation does not). Our results also show that on average our most fine-grained rules causes 33% fewer false positives than the most coarse-grained rules, and detected all of the true violations that the most coarse-grained rules can detect. This indicates rule granularity significantly affects violation detection accuracy and could be considered a tuning parameter for other rule inference and violation detection tools to reduce false positives.

On the other hand, coarse-grained rules help us discover high level rules that are shared by different security check functions, which fine-grained rules fail to uncover (examples shown in Section 5.3). These results call for attention that different levels of granularity have measurable advantages and disadvantages, and one could quantitatively evaluate the tradeoffs when designing rule inference and violation detection tools in order to choose the most suitable granularity.

In summary, AutoISES closes an important gap in achieving secure software systems. To have truly secure software systems, not only must one have a secure design, but the implementation must faithfully realize the design. To verify that the implementation faithfully realizes the design, one must write a correct code-level specification which can be verified by automatic tools such as a model checker or a static analyzer. AutoISES allows the security specifications to be automatically extracted from the actual implementation, alleviating the developers from the burden of manually writing specifications while at the same time significantly improving the accuracy of the specification.

### 1.3 Paper Layout

The remainder of the paper is organized as follows. Section 2 provides background information about MAC, DAC and the assumptions we make in our work. In Section 3, we present an overview of our approach, including some formal definitions and how we quantitatively evaluate rule granularity, followed by a detailed design in Section 4. Our methodology and experimental results are described in Section 5, and Section 6 discusses and summarizes our key techniques, their generalization and limitations. In Section 7 a discussion of the related work is presented, and finally we conclude with Section 8.

## 2 Background and Assumptions

### 2.1 DAC and MAC Background

The traditional Linux kernel uses Discretionary Access Control (DAC), meaning the access control policies are set at the discretion of the owner of the objects. For example, the root user typically sets the password file to be writable only by herself. However, if the root user mistakenly makes the password file publicly writable, then the whole system is at risk. This example shows one major deficiency of DAC, that is, mistakes of *individual policy decisions* can result in the breach of security for the *entire* system. Mandatory Access Control (MAC) is proposed to address this issue. In a MAC system, there exists a system wide security policy, such as "high-integrity file must not be modified by low-integrity users". Even if the root mistakenly grants write permissions on the password file to everyone, when a normal user tries to write the password file the attempt would fail because it is against the system-wide policy. MAC is considerably "safer" than DAC, but it is also more complex and more difficult to implement, especially for large systems like Linux [18]. It took Linux developers about two years to add MAC to the Linux kernel, and since then it has undergone many rounds of refinements and extensions. It is expected that its development will continue well into the future.

## 2.2 Assumptions

We make the following assumptions in our work.

**Reasonably mature code base:** Similar to previous work on automatic rule extraction [3, 10, 11, 20, 22], we assume that the code we work with is reasonably mature, i.e., it is mostly correct and already contains an implementation of the security architecture that is mostly working. This does not mean that software development ceases. In fact, the software might still be under active development and new features continue to be added. Almost all open source and proprietary software falls into this category, therefore this assumption does not significantly limit the applicability of this work.

**Software developers not adversarial:** We assume that software developers are trusted and will not deliberately write code to defeat our rule generation mechanism. This in general holds for majority of the software that exists today, i.e., we believe that the majority of software developers intend to write correct and secure software. In the limited cases where this assumption does not hold [25], there exist static analysis techniques that can detect such malicious code [29]. However, detecting malicious code in general is challenging and remains an active open research problem.

**Kernel and hypervisor in the trusted computing base:** For the two pieces of software that we experimented with, namely, the Linux kernel and the Xen hypervisor (virtual machine monitor), we assume that both are part of the trusted computing base. Thus, the mandatory access control is in place to prevent *user level or guest OS level* processes from breaking security policies. This assumption implies that only if a user process or a guest OS process can bypass the MAC mechanism (placed in the kernel or the hypervisor) do we consider it a breach of security. The kernel or the hypervisor is free to modify the data structures on its own behalf (e.g., for bookkeeping purposes) without going through the security checks. This assumption is adopted from the MAC architecture of the Linux kernel and Xen hypervisor.

## 3 Overview of Our Approach

In this section, we will present our high level design choices of rule inference and violation detection; formal definitions of security rules, security sensitive operations, and security violations; and how we explore different levels of rule granularity. The detailed design will be discussed in the next section.

### 3.1 Our approach

Our approach consists of two steps, as shown in Figure 3. In the first step, we generate security specifications automatically from the source code. The input to the generator is the source code and the set of security check functions. The output of this step is a set of security rules containing a security check function and a security sensitive operation represented by a group of data structure accesses as shown in Figure 2 (a) (the advantages of using a group of data structure accesses to represent a sensitive operation are discussed in Section 3.2). In the second step, AutoISES takes the source code and the rules automatically inferred in the first step as the input, and outputs ranked security violations. Note that these automatically inferred rules can be used *directly* by AutoISES without manual examination, which reduces human involvement to its minimum.

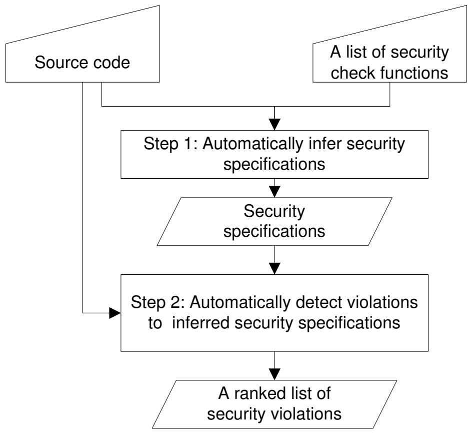

Source code
A list of security check functions
Step 1: Automatically infer security specifications
Security specifications
Step 2: Automatically detect violations to inferred security specifications
A ranked list of security violations

Figure 3: The analysis flow of AutoISES.

**Step 1: Inferring Security Rules** This paper focuses on inferring *security rules* which mandate that a security sensitive operation must be protected by a security check function, i.e., the sensitive operation must not be allowed to proceed if the security check fails. In order to effectively check such rules against the source code to detect violations, it is crucial to specify the security rule at the source code level. Unfortunately, in reality such rules are usually not documented. Therefore, our goal is to automatically infer such rules from the source code.

To infer one rule, we want to discover, for the *same* security check function, what *fixed* security sensitive operation must be protected by it. We can infer this security rule from two angles: (1) we search for all instances of *the same security check function* (e.g., `security_file_permission()`) and discover what sensitive operation is frequently protected by (e.g., performed after) the check function, or (2) we search for all instances of *the same security sensitive operation* (e.g., the 22 data structure accesses shown in Figure 2(a)) and then check what security check function is frequently used to protect (e.g., invoked before) the operation. We use the first method, because it is relatively easy to know what the security check functions are in the source code (usually documented), but knowing what security sensitive operations are in the source code itself is still a challenge (not documented). Specifically, we look for all instances of the same security check in the source code and collect sensitive operations protected by it. If this security check is frequently used to protect a fixed sensitive operation, represented by a fixed set of data structure accesses, we infer a security rule: this set of data structure accesses must be protected by this security check function. Our rationale is that released software is mostly correct, so we can infer correct behavior from it.

It is not uncommon that more than one security check function is required to protect one sensitive operation. In such cases, our inference approach still works because it will infer several separate rules, one for each security check function. The set of rules related to the same sensitive operation combined can detect violations where not all of the check functions are invoked to protect the operation.

We can infer security rules statically or dynamically. While a dynamic approach is more precise, it has poorer coverage because only executed code is analyzed. As we study large software with millions lines of code, a dynamic approach may not be sufficient, which is confirmed by previous work [9, 13]. Therefore, we use *inter-procedural* and *flow-insensitive* static program analysis for rule inference. A more detailed description of our static analysis techniques can be found in Section 4.3.

In summary, our tool AutoISES automatically infers sensitive operations in the form of a group of data structure accesses that are commonly or frequently protected by the same security check function, given a list of security check functions. Similar to previous rule inference studies [3, 10, 11, 20, 22], we cannot discover all security rules from the source code alone (discussed in Section 6.3). However, it is effective to infer some important security rules from source code, and detect previously unknown security vulnerabilities.

**Step 2: Detecting Violations** The goal of this step is to use the rules inferred above to detect security violations. Similarly, we use inter-procedural and flow-insensitive analysis for violation detection. As we already know which data structure accesses represent the security sensitive operation from an inferred rule, we can search for instances of the security sensitive operation that are not protected by the security check function, indicating security violations.

Specifically, AutoISES starts from each root function (*automatically* generated starting function for our analysis and detection as discussed later in Section 4.2.1), and collects all data structure accesses and calls to security check functions. Then it calculates the $accessViolationCount$, which is the number of accesses in the rule that are not protected by the particular security check function. Specifically, if an access in the rule is performed without being protected by a security check function, AutoISES increases the $accessViolationCount$ by one. We then use the $accessViolationCount$ for violation ranking – the higher the $accessViolationCount$ is, the more likely it is a true violation. We also allow our tool users to set up a threshold and only report violations with its $accessViolationCount$ higher than the threshold. Users can always set the threshold to zero to see all violation reports.

**Untrusted-space exposability analysis** One key technique we used to greatly reduce false positives is our *untrusted-space exposability* analysis. As we consider the kernel and the hypervisor to be our trusted computing base, security sensitive operations in kernel space and hypervisor that do not interact with the untrusted space (user space or guest OS processes), do not need to be protected by a security check function. On the other hand, if such sensitive operations interact with the untrusted space, e.g., are performed by a user space process via system calls, or use data copied from user space, then a security check may be mandatory. Since it is typical that a large number of sensitive operations are not exposed to the untrusted space, most of the detected violations would be false alarms, which is detrimental to a detection tool.

To reduce such false positives, we perform a simple trusted space exposability study. Specifically, we compiled a list of user space interface functions that are known a priori to be exposed to user space, e.g., system calls such as `sys_read()` and hypercalls. Then, AutoISES checks what sensitive operations are reachable from these interface functions. If a sensitive operation that can be exposed to the untrusted space is not protected by the proper security check function, we report the violation as *an error*; otherwise, we report the violation as *a warning*. Our goal is to ensure most of the errors are true violations, but we still generate the warnings so that developers can examine them if they want to. This approach relies on easy-to-obtain information (system calls and hypercalls) to automatically reduce the number of false positives.

## 3.2 Formal Definitions

Based on our reasoning above, we formally define the rule inferencing problem, security sensitive operations, security rules, our inference rule, and violations.

**Rule Inferencing Problem** Given the target source code and a set of $n$ kernel *security check functions*, $CheckSet = \{Check_1, ..., Check_n\}$, each of which can check if a subject (e.g., a process), is authorized to perform a certain *security sensitive operation*, $Op_i$ (e.g., read, where $1 \leq i \leq n$), on a certain object (e.g., a file) we want to uncover *security specifications* or *security rules*, $Rule_i$, in the form of a pair, ($Check_i$, $Op_i$), mandating that a security sensitive operation $Op_i$, must be *protected*, $<_{protected}$, by security check function $Check_i$ each time $Op_i$ is performed. Here protected means that the operation $Op_i$ can not be performed if the check $Check_i$ fails.

A security check function $Check_i$ can be called multiple times in the program, each of which is called an instance of the security check function, denoted as $InstanceOf(Check_i)_v$, where $v$ is between 1 and the total number of $Check_i$ instances inclusive. Similarly, a security sensitive operation $Op_i$ can appear in the program multiples times, and each of which is called an instance of the sensitive operation, $InstanceOf(Op_i)_u$. If for all instances of the sensitive operation, there exists at least one instance of security check function to protect it, then we say that the sensitive operation is protected by the security check function. Formally speaking, $\forall InstanceOf(Op_i)_u, \exists InstanceOf(Check_i)_v$, such that $InstanceOf(Op_i)_u <_{protected} InstanceOf(Check_i)_v => Op_i <_{protected} Check_i$.

**Representing Security Sensitive Operations** There are several ways to represent security sensitive operations at the code level. We can use a list of data structures that the operation manipulates, a list of functions the operation invokes, or the combination of the two. The list can be ordered or not ordered, indicating whether we require these accesses to be performed in any particular order.

**We use data structure accesses to represent a security sensitive operation, because it has two advantages** over using function calls. First, it can infer rules that function call based analysis would not be able to find. For example, if a sensitive operation is performed after a check function via different function calls, e.g., A and B, by using function A and B to represent the operation, we may mistakenly consider nothing is commonly protected by the check function and miss the rule. Zooming into the functions will allow us to find the shared data structure accesses in both A and B. Additionally, we can detect more violations by using data structure accesses. For examples, if we find that a check function always protects function call A at many places, but there is a violation that performs the same sensitive operation via function B without invoking the check function first, then we will not be able to detect the violation unless we use the data structure accesses to represent the sensitive operation in the rule. For example, `security_file_permission()` is used to protect `read`, `write`, etc., in Linux 2.6.11, but the check is missed when the sensitive operation is called through `readv` (shown in Figure 2(b)), `writev`, `aio_read`, or `aio_write`. Therefore, AutoISES would have missed all of these violations if it had not used the actual data structure accesses to represent the sensitive operation.

The tradeoffs between considering access orders or not are as follows. While preserving access orders is more precise, it has two major disadvantages. First, the order does not matter for certain rules, and preserving the order can cause one to miss the rule. For example, an directory removal operation involves setting the inode's size to 0 and decrement the number of links to it by one. The order in which the two accesses are performed is irrelevant. Second, it is more expensive to consider access orders, which can affect the scalability of our tool. On the other hand, the downside of not considering orders is that we can potentially have a higher number of false positives due to over-generalization. However, we did not find any false positives caused by this reason in this study.

Therefore, we use a set of unordered data structure accesses, $AccessSet = \{Access_1, ..., Access_m\}$, to represent sensitive operation $Op$, where each data structure access is defined as shown in Figure 4.

$$\boxed{\begin{array}{l} Access_i := READ\ AST \mid WRITE\ AST \mid ACCESS\ AST \\ AST := type\_name(->field) * \mid global\ variable \end{array}}$$

Figure 4: Definition of one data structure access. $ACCESS\ AST$ means an access to AST (Abstract Syntax Tree), either READ or WRITE.

**Security Rules** Replacing the security sensitive operation $Op_i$ with $AccessSet$ as defined above, we have the following definition of security rules:

$$Rule_i = (Check_i, AccessSet_i),\ where\, Check_i \in CheckSet$$
$$=> AccessSet_i <_{protected} Check_i.$$

**Inference Rule** As such rules are usually undocumented, we want to automatically infer them from source code by observing what sensitive operation is frequently protected by a security check function, i.e., what sensitive operation are commonly protected by different instances of the same security check function.

Formally speaking, we use the following inference rule to infer security rules:

$$AccessSet_i <_{frequently\ protected} Check_i$$

$$=> InferredRule_i = (Check_i, AccessSet_i),$$
$$where\ Check_i \in CheckSet.$$

**Violations** Using such inferred rules, we want to detect security violations. An instance of a security sensitive operation, $InstanceOf(AccessSet_i)_u$ is a violation to $InferredRule_i$ if it is not protected by any instance of the security check function. In other words,

$$Given\ InferredRule_i = (Check_i, AccessSet_i),$$

$$\forall\ InstanceOf(Check_i)_v,$$
$$InstanceOf(AccessSet_i)_u \not<_{protected} InstanceOf(Check_i)_v$$
$$=> InstanceOf(AccessSet_i)_u \in Violation_i.$$

In this paper, we use rules and inferred rules interchangeably.

## 3.3 Exploring Rule Granularity

We explore 4 different levels of granularity based on two metrics, whether to distinguish read and write access types, and whether to distinguish structure fields. The four different levels of granularity are as shown in Table 1. For example, the access `READ inode->i_size` is represented as `READ inode` for Granularity($F-$, $A+$), `ACCESS inode->i_size` for Granularity($F+$, $A-$), and `ACCESS inode` for Granularity($F-$, $A-$).

| | | Distinguishing Structure Fields | |
|---|---|---|---|
| | | Yes | No |
| Distinguishing Access Types | Yes | Granularity($F+$, $A+$) READ inode->i_size | Granularity($F-$, $A+$) READ inode |
| | No | Granularity($F+$, $A-$) ACCESS inode->i_size | Granularity($F-$, $A-$) ACCESS inode |

Table 1: Four Levels of Rule Granularity with Examples.

To better understand the impact of the rule granularity on rule inference and violation detection, and to gain insight on how well our default granularity (Granularity($F+$, $A+$)) performs, we quantitatively evaluate the 4 different levels of granularity on the Linux kernel and Xen. This exploration is orthogonal to our rule inference and violation detection, and can be applied to previous rule inference techniques [9, 11, 20, 22, 30].

## 4 Detailed Design of AutoISES

### 4.1 A Naive Approach

We first describe a naive approach and show why it does not work, which motivated us to explore alternatives. A naive approach is to start the analysis from the direct caller functions of a security check function, and consider all data structure accesses performed after the security check function as the protected sensitive operation. This approach does not work because it introduces obvious imprecision. For example, as shown in Figure 5, `security_inode_permission()` is called at the end of function `permission()`. If we start from function `permission()`, then no data structures are accessed after security check function `security_inode_permission()` in function `permission()`, indicating that no data structure access is protected by `security_inode_permission()`, which is clearly not true. This naive approach fails because `permission()` is not a function that actually uses the check to protect security sensitive operations. Instead, it is a wrapper function of the security check function. The function that actually uses the security check function for a permission check is `vfs_link()` shown in the leftmost box of Figure 5.

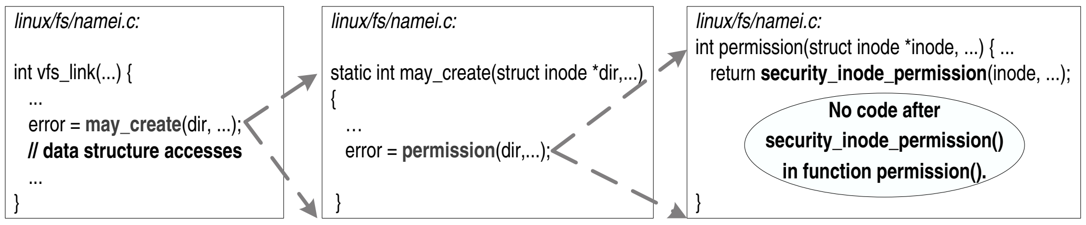
To automatically infer security rules, we need to automatically discover the functions (e.g., `vfs_link()`) that actually use security checks for authorization checking.

### 4.2 Security Specification Extraction

The goal of AutoISES is to discover the security sensitive operation, represented by a group of data structure accesses, that is protected by a security check function. Why we use data structure accesses to represent a security sensitive operation has already been discussed in Section 1.2, and the two major advantages of this representation have been described in Section 3.2. To achieve this goal, we need to address four major challenges: (1) how to automatically discover functions that actually use security checks for authorization checking; (2) how to define "protected" at the code level; (3) what information to extract; (4) how do we turn such information into security rules.

#### 4.2.1 How to find functions that actually use security checks for authorization checking?

As shown above, simply starting the analysis from the direct callers of a security check function does not work. To automatically detect security rules, we need to automatically find the functions that actually use the check function to protect sensitive operations. However, what functions actually use the check function for authorization checking depends on the semantics of the software, and thus are extremely difficult to extract automatically.



```
linux/fs/namei.c:

int vfs_link(...) {
  ...
  error = may_create(dir, ...);
  // data structure accesses
  ...
}
```

```
linux/fs/namei.c:

static int may_create(struct inode *dir,...)
{
  ...
  error = permission(dir,...);

}
```

```
linux/fs/namei.c:
int permission(struct inode *inode, ...) { ...
  return security_inode_permission(inode, ...);
}
```

Figure 5: Demonstrating the naive approach does not work.

Instead, we try to automatically extract a good approximation of these functions. Specifically, we (1) automatically break the program into modules (e.g., each file system is a module) based on the compilation configurations that come with any software (e.g. in Makefile), and (2) consider the *root functions* of each module as functions that actually use security check functions for authorization check, where *root functions* are functions that are not called by any other functions in the module. These **root functions can be automatically extracted by analyzing the call graphs** of each module.

Using this approach, AutoISES finds that `sys_link()` is a root function for the `ext2` file system module. Although `vfs_link()` is the direct user of the check, this approximation is good because the root function `sys_link()` is the caller of `vfs_link()`, therefore the root function contains all the data structure accesses `vfs_link()` performs. While it can also contain accesses that are not in `vfs_link()`, which may not be related to the security sensitive operation, it does not affect the violation detection accuracy much in practice mainly for two reasons. First, since only accesses that are protected by *many* instances of the same check function is considered as part of a sensitive operation, many unrelated accesses can be automatically eliminated during the rule generation stage (Section 4.2.4). Additionally, during the violation detection stage, we can set the threshold for $accessViolationCount$ lower to tolerate a few unrelated data structure accesses. Note that these root functions are usually a super set of our untrusted space interface functions, as many root functions can only be called by other *kernel* modules, which are considered trusted. Therefore, our untrusted-space exposability study is necessary for reducing false positives.

An alternative solution is to ask developers or tool users to provide the functions that actually use the check functions. Although it is easier to provide such functions than writing the specifications directly, it is not desirable, because (1) it is not automatic; (2) one would need to manually identify such functions each time new code is added; and (3) manually identified these functions can be error-prone.

### 4.2.2 What does "protected" mean at the code level?

An instance of a sensitive operation $Op_i$ is considered protected by an instance of a security check function $Check_i$, if the operation is allowed only if it is authorized as indicated by the return value of the check function. To implement this exact semantic, we need to know the semantics of return values of all the security check functions, which requires significant manual work and does not scale; this is not desirable. Therefore, we use a source code level approximation of this semantic: a security check function protects all data structure accesses that appear "after" the security check function in an execution trace. Although this approximation can include some unrelated accesses, it is reasonably accurate and effective at helping detecting violations for the same two reasons discussed in Section 4.2.1. Additionally, the approximation makes our approach more automatic and general, because we do not require developers to provide the semantics of the return values of the security check functions.Similar to previous static analysis techniques, our static analysis does not employ any dynamic execution information. Instead, the execution trace we use is a static approximation of the dynamic execution trace.

### 4.2.3 What information to extract?

We want to extract data structure accesses that are frequently protected by a security check function. Since a typical program accesses a large number of data structures, many of which are irrelevant to the security sensitive operation, we need to collect the most relevant accesses and exclude noise. For example, a loop iterator is not interesting for our rule extraction, so we want to exclude it. Although all data structure accesses theoretically can be protected by a security check function, structure field accesses and global variable accesses are more commonly protected than short-lived local scalar variables. Therefore, we extract all structure field accesses and global variable accesses. In addition, a security sensitive operation, being an aggregate representation of its specific instances, is naturally represented by accesses to the *types* of data structures, and not by accesses to *specific* data objects. Thus, our rule inference engine considers structure types as opposed to actual objects.

### 4.2.4 How to infer rules?

Starting from the automatically identified root functions, we can extract the data structure access set for *each instance* of a security check function. To obtain the data structure access set protected by the security check function, we simply compute the *intersection* of all of these access sets. Since our static analysis can miss some data structure accesses for some root functions due to analysis imprecision, we do not require accesses to be protected by all instances. Instead, if intersecting an access set results in an empty set, we drop this access set because it is likely to be an incomplete set. As long as there are enough security check instances protecting the accesses, we are confident the accesses are security sensitive and the inferred rule is valid.

However, different from inferring general program rules, many security check functions are called only once or twice, which makes it difficult for the intersection strategy to be effective. We observed that many such functions are only called once or twice because Linux uses a centralized place to invoke such checks for different implementations. For example, check function `security_inode_rmdir()` is only called once in the virtual file system level, but it actually protects the sensitive rmdir operation of many different file systems. Therefore, semantically the check function is invoked once for *each* file system. Thus, we can intersect the rmdir operations of different file systems to obtain the essential protected sensitive accesses. This strategy makes it possible for AutoISES to automatically generate rules of reasonably small sizes with high confidence even for check functions that are called only a few times. This is realized by performing a function alias analysis and generating a separate static trace for each function alias, essentially treating each function alias as if it was a separate function call.

## 4.3 Our Static Analysis

We use *inter-procedural* and *flow-insensitive* static program analysis to infer security rules and detect violations. It is important to use inter-procedural analysis, because many sensitive data structure accesses related to the same sensitive operation are performed in different functions. In fact, these accesses can be many (e.g., 18) levels apart in the call chain, meaning the caller of one access can be the 18th ancestor caller of another access. An intra-procedural analysis would not adequately capture the security rules or be effective at detecting violations. In fact, without our inter-procedural analysis, we would not be able to detect almost any of the violations. For higher accuracy, we perform full inter-procedural analysis, which means that we allow our analysis tool to zoom into functions as deep as it can, i.e., until it has analyzed all reachable functions whose source code is available. We chose to use flow-insensitive analysis over flow-sensitive analysis because it is less expensive and scales better for large software.

As function pointers are widely used in Linux and Xen, we perform simple function pointer analysis by resolving a function pointer to functions with the same type. Our analysis is conservative in the absence of type cast.

## 5 Methodology and Results

We evaluated our tool on the latest versions, at the time of writing, of two large open source software, Linux and Xen. Table 2 lists their size information.

| Software | Lines of Code | Total # of Check Functions |
|---|---|---|
| Linux | 5.0M | 96 |
| Xen | 0.3M | 67 |

Table 2: Evaluated software. We excluded constructor and destructor type of security check functions from the list, because they are not authorization checks.

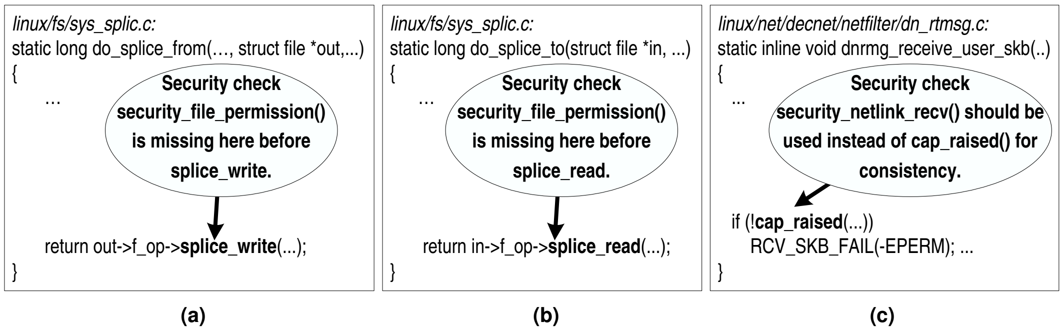
Table 3 shows our overall analysis and detection results. AutoISES automatically generated 84 code-level security rules, which served as the concrete security specifications of the two software we studied. These specifications are critical for verifying software correctness and security. Additionally they can help developers better understand the code and ease the task of software maintenance. We did not generate one rule for each security check mainly because some parts of the source code were not compiled for the default Linux kernel or Xen configuration, and were therefore not analyzed.

Based on our untrusted-space exposability study results, AutoISES reports violations that can be exposed to untrusted space as errors, and the others as warnings since they are less likely to be true security violations. Using the 84 automatically generated rules, AutoISES reported 8 error reports and 293 warning reports. A total of 8 true violations were found, 6 of which were from the error reports, and 2 were from the warnings reports (only the top warnings were examined). Among the 8 true violations, 7 of them have been confirmed by the corresponding developers. All of the automatically inferred rules were used by the AutoISES checker *directly* without being examined by us or the developers. If higher detection accuracy is desired, developers or tool users can examine rules before using them for violation detection.

These results demonstrate that AutoISES is effective at automatically inferring security rules and detecting violations to these rules, which closes an important gap in achieving security systems and greatly improves the practicality of security property checking and verification tools.

| Software | # of rules | # of True Violations | False Positives in Errors | # of Warnings | |
|---|---|---|---|---|---|
| | | | | # Inspected | Uninspected |
| Linux | 51 | 7 | 1/6 | 25 (2) | 265 |
| Xen | 33 | 1 | 1/2 | 3 | 0 |
| **Total** | **84** | **8** | **2/8** | 28 (2) | 265 |

Table 3: Overall results of AutoISES. Numbers in parentheses are true violations in warning reports.


Security check security_file_permission() is missing here before splice_write.
Security check security_file_permission() is missing here before splice_read.
Security check security_netlink_recv() should be used instead of cap_raised() for consistency.
(a)
(b)
(c)

```
linux/fs/sys_splic.c:
static long do_splice_from(..., struct file *out,...)
{
    ...
    return out->f_op->splice_write(...);
}
```

```
linux/fs/sys_splic.c:
static long do_splice_to(struct file *in, ...)
{
    ...
    return in->f_op->splice_read(...);
}
```

```
linux/net/decnet/netfilter/dn_rtmsg.c:
static inline void dnrmg_receive_user_skb(..)
{
    ...
  if (!cap_raised(...))
     RCV_SKB_FAIL(-EPERM); ...
}
```

Figure 6: True violations AutoISES automatically detected in the latest versions of Linux kernel. All of these violations have already been confirmed by the Linux developers.

## 5.1 Detected Violations

We manually examined every error report and only the top warning reports (due to time constraints) to determine if a report is a true violation or a false positive.

### 5.1.1 True Violations

There are two types of true violations, exploitable violations and consistency violations.

**Exploitable Violations** Among the 8 true violations, 5 are exploitable violations. Figure 6 (a) and (b) show two exploitable violations. In Linux 2.6.21.5, security check `security_file_permission()` was missing before the file splice read and file splice write operation. Without the check, an unauthorized user could splice data from pipe to file and vice versa, which could cause permanent data loss, information leak, etc. This violation has already been confirmed by the Linux developers.

**Consistency Violations** We term the 3 remaining true violations *Consistency Violations*, meaning that although they may not be exploitable, they violate the consistency of using security check functions. Such inconsistencies can confuse developers and make the software difficult to maintain, both of which can contribute to more errors in the future. Therefore, it is important for developers to fix consistency violations.

Figure 6 (c) shows an example of a consistency violation. A security check `security_netlink_recv()`, which checks permission before processing the received netlink message, was missing in `dnrmg_receive_user_skb()`, which receives and processes netlink messages. This error could cause the kernel to receive messages from unauthorized users. However, `dnrmg_receive_user_skb()` did call function `cap_raised()`, which is what `security_netlink_recv()` calls eventually. In other words, it bypasses the security check interface functions, and calls the backend security policy functions, which is a bad practice and should be avoided.

At the time of writing, 2 out of the 3 consistency violations, including the example shown above, have been confirmed and fixed by the corresponding developers.

### 5.1.2 False Positives

The false positive rate in error reports is 2 out of 8. There are more false positives in the warning reports because no untrusted-space exposability analysis is performed on the warning reports. Developers can choose to focus on the error reports to save time, or also examine the warnings if desired.

Several factors can contribute to false positives. First, as we use conservative function pointer alias analysis, we can mistakenly consider accesses not related to an operation as part of the operation, and generate an imprecise rule. These extra accesses do not need to be protected by the security check, but our tool may still report such false violations. A static analysis tool with more advanced function pointer alias analysis could reduce such false positives.

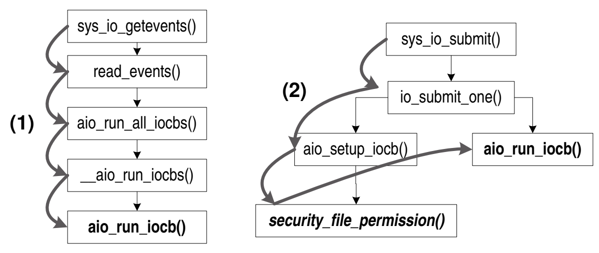


Figure 7: A false positive detected by AutoISES in Linux kernel 2.6.21.5. Only related functions are shown.

Additionally, certain semantics of the target code make some of the detected errors not exploitable. Figure 7 shows such an example where an implicit temporal constraint on certain system calls allows the coverage of a security check to span multiple system calls. AutoISES reported that a security check `security_file_permission()` should be called before `aio_run_iocb()`, but in the call chain in Figure 7(1) starting from a system call function `sys_io_getevents()`, the check `security_file_permission()` was missing. However, this is not an exploitable violation, because system call `sys_io_getevents()` cannot be called without system call `sys_io_submit()` being invoked first, which consults the proper security check in its callee function `aio_setup_iocb()` in call chain (2). Because AutoISES did not know this restriction in using the system calls, it reports the violation. However, if the file permission is changed after the setup system call `sys_io_submit()` and before the invocation of `sys_io_getevents()`, then unauthorized accesses can occur. Linux developers confirmed the potential of such violations, but are unlikely to fix it because the current Linux implementation does not enforce protection against this type of violations.

There are at least two ways to reduce or eliminate false positives. First, we can employ more accurate static analysis techniques. Additionally, as increasing granularity could reduce false positives (discussed later in Section 5.3), we can experiment with even finer granularity, such as distinguishing increment, decrement, and zeroing operations, to further reduce false positives.

## 5.2 Parameter Sensitivity and Time Overhead

By default, we set the threshold of $accessViolationCount$ to be 50% of the rule size, which is the total number of data structure accesses in a rule. We found that for Linux, the detection results are not very sensitive to this parameter, meaning that most true violations perform all or almost all of the data structure accesses, and false violations often perform none or only a few of the data structure accesses. These results show that the generated rules capture the implicit security rules well, and these rules are effective in helping detecting violations to them. For Xen, the results are more sensitive to the threshold. A possible explanation is that, in general Xen security checks are called fewer times compared to Linux kernel, therefore, there are fewer instances for AutoISES to learn precise rules. As a result, the inferred Xen rules contain more noisy accesses that do not need to be protected by the check functions. In this case, we set the threshold to be higher, 90%, to minimize the impact of noisy accesses.

AutoISES spent 86 minutes on inferring 51 rules from the entire Linux kernel, and 116 minutes on using these rules to check for violations in the entire Linux kernel. As the code size of Xen is much smaller, the time spent on Xen rule generation is 25 seconds, and 39 minutes for detection. This shows that our tool is efficient enough to be used in practice for large real world software.

## 5.3 Impact of Rule Granularity

In many cases, a coarse-grained rule is overly generalized and thus does not precisely represent the implicit security rules. For example, two different checks, `security_file_link()` and `security_file_unlink()` are designed to protect two different inode operations. However, as shown in Figure 8(a), the inferred operations of Granularity($F-$, $A+$) are the same. Using finer granularity, Granularity($F+$, $A+$), AutoISES is able to automatically infer two different operations (Figure 8(b)-(c)). For example, the unlink operation contains access `READ inode->i_size`, which is not part of the link operation.

Fine-grained rules cause less false positives during the detection stage. For 5 randomly selected security checks, compared with the most coarse-grained rules (Granularity($F-$,$A-$)) our most fine-grained rules (Granularity($F+$,$A+$)) on average cause 33% fewer false positives (in both error reports and warning reports). Granularity($F+$, $A-$) cause 20% fewer false positives, and Granularity($F-$, $A+$) 13.3% fewer. The results show that using finer granularity can greatly reduce the number of false positives, and adjusting the rule granularity could be considered as an important tuning parameter for other rule inference and violation detection tools [9, 11, 20, 22, 30].

Although coarse-grained rules produce a higher false positive rate, they can provide very useful information that fine-grained rules may fail to unveil. In the example above, the operation of Granularity($F-$, $A+$) is shared by almost all inode related security

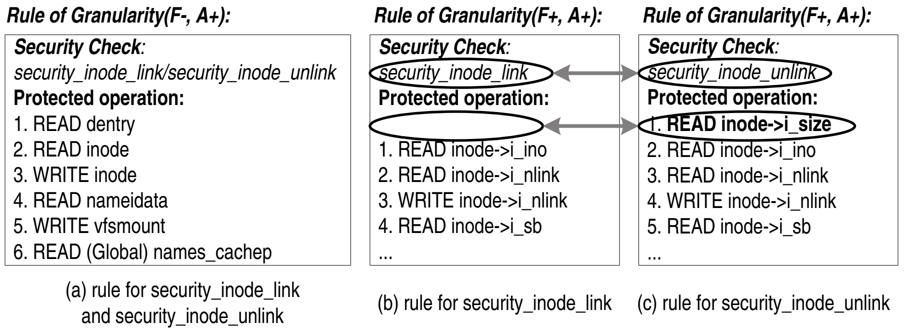


Figure 8: For two security checks, `security_file_link()` and `security_file_unlink()`, the inferred operations of Granularity($F-$, $A+$) are the same. If we use Granularity($F+$, $A+$), the learned operations are different, e.g., the unlink operation contains an extra access, `READ inode->i_size`.

check functions, including `security_inode_rename()`, `security_inode_rmdir()`, `security_inode_mkdir()`, `security_file_link()`, `security_file_unlink()`, `security_inode_symlink()`, etc. Therefore, the rule represent the common accesses of inode operations in general. A more fine-grained rule may fail to reveal the common behavior among all inode and file operations.

In addition, a fine-grained rule can be overly specific, and cause false negatives. We did not observe such cases for our most fine-grained rules in this study, i.e., our most fine-grained rules were able to detect all of the true violations. The result indicates that the default granularity we use is the best among the 4 levels of granularity in terms of detection accuracy, as they produce the least number of false positives, and the same number of false negatives as the coarse-grained rules. In the future, we plan to experiment with even finer granularity and its impact on both false positives and false negatives.

Results from different levels of granularity can be used as a metric for violation ranking. For example, a violation that is reported at all levels of granularity is probably more likely to be a true violation than one that is reported only at some levels. In our future work, we will explore using the number of granularity levels a violation occurs at to rank violations.

## 6 Discussions and Limitations

### 6.1 Key Techniques that Make AutoISES Work

Automatically generating security specifications poses several key challenges that make previous static analysis tools not directly applicable. We designed five important techniques (first four are new) to address these challenges as summarized below (Sec. column lists corresponding sections that describe the techniques):

| Challenge | Our Solution | Sec. |
|---|---|---|
| How to represent a sensitive operation at the code level | Use *Data structure accesses* based on our key observation | 1.2 & 3.2 |
| High false positive rates as many sensitive operations can not be exposed to the untrusted space | Simple untrusted space exposability study to greatly reduce false positives | 3.1 |
| Root functions for analysis: Cannot simply start analysis from direct callers of a security check function | Automatic root function discovery: *Automatically* discover functions that actually use security check functions for authorization check | 4.2.1 |
| Insufficient invocation instances of security check functions | Leverage different implementations (e.g., from different file systems) of the same operation | 4.2.4 |
| Data structure accesses are spread in different functions. | Interprocedural analysis with function pointer analysis | 4.3 |

### 6.2 Generalization

Although many of the solutions described above are designed for inferring security specifications and detecting security violations, some of the ideas are *general*, and can be applied to other applications. For example, our security rules are an important type of function-data correlation. Such function-data correlations widely exist in programs. Violating these implicit constraints results in buggy programs that may cause severe damage. Our techniques can be used to infer those general function-data correlations, e.g., a lock acquisition function required before accessing shared data structures, which can be used for detecting concurrency bugs. In addition, the strategy of using multiple implementations of the same virtual API to generate more precise rules is generally applicable to situations where source code at the virtual API level is not sufficient to generate reliable rules.

### 6.3 Limitations

**False Negatives** Similar to previous static analysis work, our approach can miss security violations. First, if a security check function is not invoked at all (e.g., `security_sk_classify_flow` is not used in Linux 2.6.11 yet) or the list of security check functions is incomplete, we would not be able to infer rules or detect violations related to these missing check functions.

Additionally, our analysis uses only data structure accesses to represent a security operation. Therefore, if the source code of such low-level accesses is not available, AutoISES will not be able to extract information about them, and the representation of the sensitive operation would be incomplete, potentially causing false negatives.

Moreover, AutoISES does not verify if the security check is performed on the same object as the sensitive operation. Therefore, if the proper security check is invoked, but on a different object, AutoISES will not detect this violation. Matching the actual object remains as our future work. Additionally, our flow-insensitive analysis could introduce false negatives. For example, if a security check is missing on a taken branch, but the check is invoked on the non-taken branch, AutoISES may not be able to detect the violation. Using a flow-sensitive analysis could address this problem.

**Difficulty in Verifying Violations** We manually examine error reports and warning reports to determine if a report is a true violation or a false positive. Unlike errors such as buffer overflows and null pointer dereferences, which are usually easy to confirm after the error is detected, the manual verification process for security violations is more difficult. To decide if a violation is exploitable, one needs to understand the semantics of the code, knowing what operations can interact with the untrusted space, such as the user space for Linux, and design a feasible way to exploit the attack. Conversely, to determine if a violation is a false positive, one needs to prove that either the operation is security insensitive, or that it is indeed covered by a security check that was not included due to analysis imprecision. Sometimes it requires deep knowledge of not only the target software, but also how the APIs are used by client software (e.g., the example discussed in Section 5.1.2). Such difficulties are mostly due to the inherent characteristics of security violations. However, we imagine that the task would be much easier for the original developers as they possess deep semantics knowledge of the code.

**Non-authorization Checks** A small number of security checks are not authorization checks, which do not protect any security operations. For example, `security_sk_free()` should be called *after* using a kernel sk buffer to clear sensitive data. Our current implementation does not support such rules where a security check function must be invoked after a certain operation. However, such rules can be easily supported by extending our current implementation to include the post-operation checks.

## 7 Related Work

**Mining Security Sensitive Operations** Ganapathy et al. used concept analysis to find fingerprints of security sensitive operations [15]. While both this approach and AutoISES try to map the high level security sensitive operation (e.g., rmdir) to its implementation (e.g., the C code sequences that actually perform the remove directory operation), there are two major differences. First, the goals and assumptions are different. We aim to identify the pairing relationship between a security check and the code level representation of the sensitive operation that the check guards. Thus we assume the code already implements a reference monitor and is mostly correct; our goal is therefore to discover cases where the reference monitor is bypassed. Ganapathy's goal, on the other hand, is to retrofit code with security. Thus they assume that the code does not have security built in. Rather, they need to identify sequences of code that represent a unit of security sensitive operation and that should be guarded by a security hook. In order to do that they need more prior knowledge with regard to the API and the security sensitive data structures. In our case, all information except the list of security check functions, and the list of system call functions and hypercall functions, is inferred from the code itself. Second, while our inferred operations are used directly by our checker without being examined manually, their operations still require manual refinement prior to use.

Although automatic hook placement is promising, it has not been adopted in reality yet. Therefore, while we should encourage automatic hook placement, it is still highly desirable to seek alternative, complementary solutions that can automatically infer security rules from existing or legacy source code and detect security vulnerabilities.

**Detection and Verification Tools** The past years have seen a proliferation of program analysis and verification tools that can be used to detect security vulnerabilities or verify security properties [2, 4, 5, 6, 9, 12, 14, 16, 18, 27, 30]. However, no previous work can automatically generate code-level security specifications and instead require developers or users to provide these specifications. Previous work [30] takes *manually* identified simple security rules to check for security vulnerabilities. As discussed in details in Section 1, the rules are coarse and imprecise, resulting in many false alarms. Additionally, the approach can potentially fail to detect cases where the check and the operation does not match because the rules do not specify which check is required for which operation. Edwards et al. dynamically detect inconsistencies between data structure accesses to identify security vulnerabilities [9]. While a dynamic approach is generally more accurate, it suffers from coverage problem - only code that is executed can be analyzed. In addition, it requires manually written filtering rules to guide the trace analysis in order to detect security violations.

**Inferring Programming Rules** Several techniques have been proposed to infer different types of programming rules from source code or execution trace [3, 10, 11, 20, 22, 24]. As already discussed in Section 1, previous techniques is not directly applicable to our problem, because they are limited by the types of rules they can infer. Specifically, Engler et al. extract programming rules based on several manually identified rule templates, such as function $\langle A \rangle$ and $\langle B \rangle$ should be paired, function $\langle F \rangle$ must be checked for failure, and null pointer $\langle P \rangle$ should not be dereferenced [10]. PR-Miner focuses on inferring correlations among *functions* [20]. Variable value related program invariants are inferred by Daikon [11], and MUVI[22] infers *variable-variable* correlations for detecting multi-variable inconsistent update bugs and multi-variable concurrency bugs. A few other approaches infer API and/or abstract data type related rules[3, 24]. Different from all these studies, we infer rules related to security *functions protecting a group of data structure accesses* based on our key observation. Inferring different types of rules requires different techniques. In addition, dynamic analysis is used in [3, 11], therefore the coverage is limited because only instrumented and executed code is used for rule learning. Moreover, unlike PR-Miner which uses only intraprocedural analysis, our analysis is interprocedural, which is one of the key techniques that allow us to infer complicated and detailed security rules. Additionally, while PR-Miner uses more complex data mining techniques to infer programming rules, we leverage readily available prior knowledge about part of our rules, the security check functions, so that we can extract security rules without expensive data mining techniques.

**Inferring Models and Rules in General** The general idea of automatically extracting models from low-level implementation has been discussed in previous literature [8, 17, 21]. For example, Lie et al. proposed automatic extraction of specifications from actual protocol code and then running the extracted specifications through a model checker [21]. While conceptually these approaches bear some resemblance to the approach taken by AutoISES, we are the first to show the feasibility of automatic extraction of security specifications from actual implementation. In addition, none of the previous tools have demonstrated the ability to scale to programs the size of the Linux kernel.

Lee et al. [19] use data mining techniques to learn intrusion detection model for adaptive intrusion detection. Tongaonkar et al. [26] infer high-level security policy from low level firewall filtering rules. None of these work infer access control related security rules.

## 8 Conclusions and Future Work

This paper makes two contributions. One is to automatically infer code-level security rules and detect security violations. Our tool, AutoISES, automatically inferred 84 security rules from the latest versions of Linux kernel and Xen, and used them to detect 8 security vulnerabilities, demonstrating the effectiveness of our approach. The second contribution is to take the first step to quantitatively study the impact of the rule granularity on rule generation and verification. This approach is orthogonal to our first contribution, and can be applied to other rule inference tools.

While this work focuses on rule inference and violation detection in Linux kernel and Xen, our techniques can be used to generate rules and detect violations in other access control systems. In addition, the techniques can be applied to infer general function-data correlation type of rules, such as lock acquisition functions protecting shared variables accesses. In the future, we plan to improve our analysis and detection accuracy by employing a more advanced static analysis tool and using finer rule granularity.

## 9 Acknowledgments

We greatly appreciate the anonymous reviewers for their feedback. We thank Mihai Christodorescu, Brett Daniel, Darko Marinov and Vugranam Sreedhar for their invaluable comments and observations during this work. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants.

## References

[1] Vulnerability summary CVE-2006-1856. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-1856.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, 2007.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, 2002.

[4] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proceedings of the 11th International Static Analysis Symposium*, 2004.

[6] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[7] G. S. Coker. Xen security modules: Intro. http://lists.xensource.com/archives/html/xense-devel/2006-09/msg00000.html.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[9] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[10] D. R. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[12] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[13] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[14] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[15] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[17] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. *Software Testing, Verification and Reliability*, 2001.

[18] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the linux security modules framework. *ACM Transactions on Information and System Security*, 2004.

[19] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 2000.

[20] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[21] D. Lie, A. Chou, D. Engler, and D. L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[22] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[23] NSA. Security-Enhanced Linux (SELinux). Available at http://www.nsa.gov/selinux.

[24] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 174–184.

[25] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 1995.

[26] A. Tongaonkar, N. Inamdar, and R. Sekar. Inferring higher level policies from firewall rules. In *Proceedings of the 21st Large Installation System Administration Conference*, 2007.

[27] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 2000 Network and Distributed Systems Security Conference*, 2000.

[28] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[29] C. Wysopal and C. Eng. Static detection of application backdoors. Available at http://www.veracode.com/images/stories/static-detection-of-backdoors-1.0.pdf.

[30] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

## Notes

[1]Portions of the work performed while Lin Tan was a summer intern at IBM Research.

# Real-World Buffer Overflow Protection for Userspace & Kernelspace

Michael Dalton, Hari Kannan, Christos Kozyrakis
*Computer Systems Laboratory*
*Stanford University*
$\{$*mwdalton, hkannan, kozyraki*$\}$*@stanford.edu*

## Abstract

Despite having been around for more than 25 years, buffer overflow attacks are still a major security threat for deployed software. Existing techniques for buffer overflow detection provide partial protection at best as they detect limited cases, suffer from many false positives, require source code access, or introduce large performance overheads. Moreover, none of these techniques are easily applicable to the operating system kernel.

This paper presents a practical security environment for buffer overflow detection in userspace and kernelspace code. Our techniques build upon dynamic information flow tracking (DIFT) and prevent the attacker from overwriting pointers in the application or operating system. Unlike previous work, our technique does not have false positives on unmodified binaries, protects both data and control pointers, and allows for practical hardware support. Moreover, it is applicable to the kernel and provides robust detection of buffer overflows and user/kernel pointer dereferences. Using a full system prototype of a Linux workstation (hardware and software), we demonstrate our security approach in practice and discuss the major challenges for robust buffer overflow protection in real-world software.

## 1 Introduction

Buffer overflows remain one of the most critical threats to systems security, although they have been prevalent for over 25 years. Successful exploitation of a buffer overflow attack often results in arbitrary code execution, and complete control of the vulnerable application. Many of the most damaging worms and viruses [8, 27] use buffer overflow attacks. Kernel buffer overflows are especially potent as they can override any protection mechanisms, such as Solaris jails or SELinux access controls. Remotely exploitable buffer overflows have been found in modern operating systems including Linux [23], Windows XP and Vista [48], and OpenBSD [33].

Despite decades of research, the available buffer overflow protection mechanisms are partial at best. These mechanisms provide protection only in limited situations [9], require source code access [51], cause false positives in real-world programs [28, 34], can be defeated by brute force [44], or result in high runtime overheads [29]. Additionally, there is no practical mechanism to protect the OS kernel from buffer overflows or unsafe user pointer dereferences.

Recent research has established *dynamic information flow tracking (DIFT)* as a promising platform for detecting a wide range of security attacks on unmodified binaries. The idea behind DIFT is to tag (taint) untrusted data and track its propagation through the system. DIFT associates a tag with every memory location in the system. Any new data derived from untrusted data is also tagged. If tainted data is used in a potentially unsafe manner, such as dereferencing a tagged pointer, a security exception is raised. The generality of the DIFT model has led to the development of several software [31, 32, 38, 51] and hardware [5, 10, 13] implementations.

Current DIFT systems use a security policy based on *bounds-check recognition (BR)* in order to detect buffer overflows. Under this scheme, tainted information must receive a bounds check before it can be safely dereferenced as a pointer. While this technique has been used to defeat several exploits [5, 10, 10, 13], it suffers from many false positives and false negatives. In practice, bounds checks are ambiguously defined at best, and may be completely omitted in perfectly safe situations [12, 13]. Thus, the applicability of a BR-based scheme is limited, rendering it hard to deploy.

Recent work has proposed a new approach for preventing buffer overflows using DIFT [19]. This novel technique prevents *pointer injection (PI)* by the attacker. Most buffer overflow attacks are exploited by corrupting and overwriting legitimate application pointers. This technique prevents such pointer corruption and does not rely on recognizing bounds checks, avoiding the false positives associated with BR-based analyses. However, this work has never been applied to a large application, or the operating system kernel. Moreover, this technique requires hardware that is extremely complex and impractical to build.

This paper presents a practical approach for preventing buffer overflows in userspace and kernelspace using a pointer injection-based DIFT analysis. Our approach identifies and tracks all legitimate pointers in the application. Untrusted input must be combined with a legitimate pointer before being dereferenced. Failure to do so will result in a security exception.

The specific contributions of this work are:

- We present the *first* DIFT policy for buffer overflow prevention that runs on stripped, unmodified binaries, protects both code and data pointers, and runs on real-world applications such as GCC and Apache without false positives.

- We demonstrate that the same policy is applicable to the Linux kernel. It is the *first* security policy to dynamically protect the kernel code from buffer overflows and user-kernel pointer dereferences without introducing false positives.

- We use a *full-system DIFT prototype* based on the SPARC V8 processor to demonstrate the integration of hardware and software techniques for robust protection against buffer overflows. Our results are evaluated on a Gentoo Linux platform. We show that hardware support requirements are reasonable and that the performance overhead is minimal.

- We discuss practical shortcomings of our approach, and discuss how flaws can be mitigated using additional security policies based on DIFT.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 summarizes the Raksha architecture and our full-system prototype. Section 4 presents our policy for buffer overflow protection for userspace applications, while Section 5 extends the protection to the operating system kernel. Section 6 discusses weaknesses in our approach, and how they can be mitigated with other buffer overflow prevention policies. Finally, Section 7 concludes the paper.

## 2 Related Work

Buffer overflow prevention is an active area of research with decades of history. This section summarizes the state of the art in buffer overflow prevention and the shortcomings of currently available approaches.

### 2.1 Existing Buffer Overflow Solutions

Many solutions have been proposed to prevent pointer or code corruption by untrusted data. Unfortunately, the solutions deployed in existing systems have drawbacks that prevent them from providing comprehensive protection against buffer overflows.

**Canary-based buffer overflow protection** uses a random, word-sized canary value to detect overwrites of protected data. Canaries are placed before the beginning of protected data and the value of the canary is verified each time protected data is used. A standard buffer overflow attack will change the canary value before overwriting protected data, and thus canary checks provide buffer overflow detection. Software implementations of canaries typically require source code access and have been used to protect stack linking information [16] and heap chunk metadata [39]. Related hardware canary implementations [21, 46] have been proposed to protect the stack return address and work with unmodified binaries.

However, buffer overflows may be exploited without overwriting canary values in many situations. For example, in a system with stack canaries, buffer overflows may overwrite local variables, even function pointers, because the stack canary only protects stack linking information. Similarly, heap overflows can overwrite neighboring variables in the same heap chunk without overwriting canaries. Additionally, this technique may change data structure layout by inserting canary words, breaking compatibility with legacy applications. Canary-based approaches also do not protect other memory regions such as the global data segment, BSS or custom heap allocation arenas.

**Non-executable data protection** prevents stack or heap data from being executed as code. Modern hardware platforms, including the x86, support this technique by enforcing executable permissions on a per-page basis. However, this approach breaks backwards compatibility with legacy applications that generate code at runtime on the heap or stack. More importantly, this approach only prevents buffer overflow exploits that rely on code injection. Rather than injecting new code, attackers can take control of an application by using existing code in the application or libraries. This form of attack, known as a *return-into-libc* exploit, can perform arbitrary computations and in practice is just as powerful as a code injection attack [43].

**Address space layout randomization (ASLR)** is a buffer overflow defense that randomizes the memory locations of system components [34]. In a system with ASLR, the base address of each memory region (stack, executable, libraries, heap) is randomized at startup. A standard buffer overflow attack will not work reliably, as the security-critical information is not easy to locate in memory. ASLR has been adopted on both Linux and Windows platforms. However, ASLR is not backwards compatible with legacy code, as it requires programs to be recompiled into position-independent executables [49] and will break code that makes assumptions about memory layout. ASLR must be disabled for the entire process if it is not supported by the executable or any shared libraries. Real-world exploits such as the Macromedia Flash buffer overflow attack [14] on Windows Vista have trivially bypassed ASLR because the vulnerable application or its third-party libraries did not have ASLR support.

Moreover, attackers can easily circumvent ASLR on 32-bit systems using brute-force techniques [44]. On little-endian architectures such as the x86, partial overwrite attacks on the least significant bytes of a pointer have been used to bypass ASLR protection [3, 17]. Additionally, ASLR implementations can be compromised if pointer values are leaked to the attacker by techniques such as format string attacks [3].

Overall, while existing defense mechanisms have raised the bar, buffer overflow attacks remain a problem. Real-world exploits such as [14] and [17] demonstrated that a seasoned attacker can bypass even the combination of ASLR, stack canaries, and non-executable pages.

## 2.2 Dynamic Information Flow Tracking

Dynamic Information Flow Tracking (DIFT) is a practical platform for preventing a wide range of security attacks from memory corruptions to SQL injections. DIFT associates a tag with every memory word or byte. The tag is used to taint data from untrusted sources. Most operations propagate tags from source operands to destination operands. If tagged data is used in unsafe ways, such as dereferencing a tainted pointer or executing a tainted SQL command, a security exception is raised.

DIFT has several advantages as a security mechanism. DIFT analyses can be applied to unmodified binaries. Using hardware support, DIFT has negligible overhead and works correctly with all types of legacy applications, even those with multithreading and self-modifying code [7, 13]. DIFT can potentially provide a solution to the buffer overflow problem that protects all pointers (code and data), has no false positives, requires no source code access, and works with unmodified legacy binaries and even the operating system. Previous hardware approaches protect only the stack return address [21, 46] or prevent code injection with non-executable pages.

There are two major policies for buffer overflow protection using DIFT: *bounds-check recognition (BR)* and *pointer injection (PI)*. The approaches differ in tag propagation rules, the conditions that indicate an attack, and whether tagged input can ever be validated by application code.

Most DIFT systems use a BR policy to prevent buffer overflow attacks [5, 10, 13, 38]. This technique forbids dereferences of untrusted information without a preceding bounds check. A buffer overflow is detected when a tagged code or data pointer is used. Certain instructions, such as logical AND and comparison against constants, are assumed to be bounds check operations that represent validation of untrusted input by the program code. Hence, these instructions untaint any tainted operands.

Unfortunately, the BR policy leads to significant *false negatives* [13, 19]. Not all comparisons are bounds checks. For example, the glibc $strtok()$ function compares each input character against a class of allowed characters, and stores matches in an output buffer. DIFT interprets these comparisons as bounds checks, and thus the output buffer is always untainted, even if the input to $strtok()$ was tainted. This can lead to false negatives such as failure to detect a malicious return address overwrite in the atphttpd stack overflow [1].

However, the most critical flaw of BR-based policies is an unacceptable number of *false positives* with commonly used software. Any scheme for input validation on binaries has an inherent false positive risk. While the tainted value that is bounds checked is untainted, none of the aliases for that value in memory or other registers will be validated. Moreover, even trivial programs can cause false positives because not all untrusted pointer dereferences need to be bounds checked [13]. Many common glibc functions, such as $tolower()$, $toupper()$, and various character classification functions ($isalpha()$, $isalnum()$, etc.) index an untrusted byte into a 256 entry table. This is completely safe, and requires no bounds check. However, BR policies fail to recognize this input validation case because the bounds of the table are not known in a stripped binary. Hence, false positives occur during common system operations such as compiling files with gcc and compressing data with gzip. In practice, false positives occur only for data pointer protection. No false positive has been reported on x86 Linux systems so long as only control pointers are protected [10]. Unfortunately, control pointer protection alone has been shown to be insufficient [6].

Recent work [19] has proposed a pointer injection (PI) policy for buffer overflow protection using DIFT. Rather than recognize bounds checks, PI enforces a different invariant: untrusted information should never directly supply a pointer value. Instead, tainted information must always be combined with a legitimate pointer from the application before it can be dereferenced. Applications frequently add an untrusted index to a legitimate base address pointer from the application's address space. On the other hand, existing exploitation techniques rely on injecting pointer values directly, such as by overwriting the return address, frame pointers, global offset table entries, or malloc chunk header pointers.

To prevent buffer overflows, a PI policy uses two tag bits per memory location: one to identify tainted data (T bit) and the other to identify pointers (P bit). As in other DIFT analyses, the taint bit is set for all untrusted information, and propagated during data movement, arithmetic, and logical instructions. However, PI provides no method for untainting data, nor does it rely on any bounds check recognition. The P bit is set only for legitimate pointers in the application and propagated only during valid pointer operations such as adding a pointer to a non-pointer or aligning a pointer to a power-of-2 boundary. Security attacks are detected if a tainted pointer is dereferenced and the P bit is not set. The primary advantage of PI is that it does not rely on bounds check recognition, thus avoiding the false positive and negative issues that plagued the BR-based policies.

The disadvantage of the PI policy is that it requires legitimate application pointers to be identified. For dynamically allocated memory, this can be accomplished by setting the P bit of any pointer returned by a memory-allocating system call such as $mmap$ or $brk$. However, no such solution has been presented for pointers to statically allocated memory regions. The original proposal requires that each `add` or `sub` instruction determines if one of its untainted operands points into any valid virtual address range [19] . If so, the destination operand has its P bit set, even if the source operand does not. To support such functionality, the hardware would need to traverse the entire page table or some other variable length data-structure that summarizes the allocated portions of the virtual address space for every add or subtract instruction in the program. The complexity and runtime overhead of such hardware is far beyond what is acceptable in modern systems. Furthermore, while promising, the PI policy has not been evaluated on a wide range of large applications, as the original proposal was limited to simulation studies with performance benchmarks.

DIFT has never been used to provide buffer overflow protection for the operating system code itself. The OS code is as vulnerable to buffer overflows as user code, and several such attacks have been documented [23, 33, 48]. Moreover, the complexity of the OS code represents a good benchmark for the robustness of a security policy, especially with respect to false positives.

## 3 DIFT System Overview

Our experiments are based on Raksha, a full-system prototype with hardware support for DIFT [13]. Hardware-assisted DIFT provides a number of advantages over software approaches. Software DIFT relies on dynamic binary translation and incurs significant overheads ranging from 3× to 37× [31, 38]. Software DIFT does not work with self-modifying code and leads to races for multithreaded code that result in false positives and negatives [7]. Hardware support addresses these shortcomings, and allows us to apply DIFT analysis to the operating system code as well.

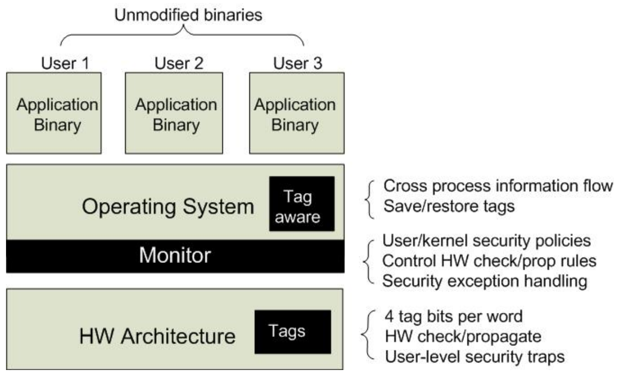


Figure 1: The system stack for the Raksha DIFT platform.

### 3.1 The Raksha Architecture

Raksha is a DIFT platform that includes hardware and software components. It was the first DIFT system to prevent both high-level attacks such as SQL injection and cross-site scripting, and lower-level attacks such as format strings and buffer overflows on unmodified binaries [13]. Prior to this work, Raksha only supported a BR-based policy for buffer overflow protection, and encountered the associated false positives and negatives.

Raksha extends each register and memory word by four tag bits in hardware. Each bit supports an independent security policy specified by software using a set of *policy configuration registers* that define the rules for propagating and checking the tag bits for untrusted data. Tags and configuration registers are completely transparent to applications, which are unmodified binaries.

**Tag operations:** Hardware is extended to perform tag propagation and checks in addition to the functionality defined by each instruction. All instructions in the instruction set are decomposed into one or more *primitive operations* such as arithmetic, logical, etc. Check and propagate rules are specified by software at the granularity of primitive operations. This allows the security policy configuration to be independent of instruction set complexity (CISC vs RISC), as all instructions are viewed as a sequence of one or more primitive operations. For example, the subtract-and-compare instruction in the SPARC architecture is decomposed into an arithmetic operation and a comparison operation. Hardware first performs tag propagation and checks for the arithmetic operation, followed by propagation and checks for the comparison operation. In addition, Raksha allows software to specify custom rules for a small number of individual instructions. This enables handling corner cases within a primitive operation class. For example, "*xor r1,r1,r1*" is a commonly used idiom to reset registers, especially on x86 machines. Software can indicate that such an instruction untaints its output operand.

The original Raksha hardware supported AND and OR propagation modes when the tag information of two input operands is combined to generate the tag for the output operand. For this work, we found it necessary to support a logical XOR mode for certain operations that clear the output tag if both tags are set.

The Raksha hardware implements tags at word granularity. To handle byte or halfword updates, the propagation rules can specify how to merge the new tag from the partial update with the existing tag for the whole word. Software can also be used to maintain accurate tags at byte granularity, building upon the low overhead exception mechanism listed below. Nevertheless, this capability was not necessary for this work, as we focus on protecting pointers which are required to be aligned at word boundaries by modern executable file formats [41].

**Security Exceptions:** Failing tag checks result in security exceptions. These exceptions are implemented as *user-level exceptions* and incur overhead similar to that of a function call. As security exceptions do not require a change in privilege level, the security policies can also be applied to the operating system. A special *trusted mode* provides the security exception handler with direct access to tag bits and configuration registers. All code outside the handler (application or OS code) runs in untrusted mode, and may not access tags or configuration registers. We prevent untrusted code from accessing, modifying, or executing the handler code or data by using one of the four available tag bits to implement a sandboxing policy that prevents loads and stores from untrusted code to reference monitor memory [13]. This ensures handler integrity even during a memory corruption attack on the application.

At the software level, Raksha introduces a security monitor module. The monitor is responsible for setting the hardware configuration registers for check and propagate rules based on the active security policies in the system. It also includes the handler that is invoked on security exceptions. While in some cases a security exception leads to immediate program termination, in other cases the monitor invokes additional software modules for further processing of the security issues. For example, SQL injection protection raises a security exception on every database query operation so that the security monitor may inspect the current SQL query and verify that it does not contain a tainted SQL command.

## 3.2 System Prototype

Figure 1 provides an overview of the Raksha system along with the changes made to hardware and software components. The hardware is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [22]. We modified Leon to include the security features of Raksha and mapped it to a Virtex-II Pro FPGA. Leon uses a single-issue, 7-stage pipeline with first-level caches. Its RTL code was modified to add 4-bit tags to all user-visible registers, and cache and memory locations. In addition, Raksha's configuration and exception registers, as well as instructions that directly access tags and manipulate the special registers, were added to the Leon. Overall, we added 9 instructions and 16 registers to the SPARC V8 ISA.

The resulting system is a *full-featured SPARC Linux workstation* running Gentoo Linux with a 2.6 kernel. DIFT policies are applied to all userspace applications, which are unmodified binaries with no source code access or debugging information. The security framework is extensible through software, can track information flow across address spaces, and can thwart attacks employing multiple processes. Since tag propagation and checks occur in hardware and are parallel with instruction execution, Raksha has minimal impact on the observed performance [13].

Although the following sections will discuss primarily the hardware and software issues we observed with the SPARC-based prototype, we also comment on the additional issues, differences, and solutions for other architectures such as the x86.

## 4 BOF Protection for Userspace

To provide comprehensive protection against buffer overflows for userspace applications, we use DIFT with a pointer injection (PI) policy. In contrast to previous work [19], our PI policy has no false positives on large Unix applications, provides reliable identification of pointers to statically allocated memory, and requires simple hardware support well within the capabilities of proposed DIFT architectures such as Raksha.

## 4.1 Rules for DIFT Propagation & Checks

Tables 1 and 2 present the DIFT rules for tag propagation and checks for buffer overflow prevention. The rules are intended to be as conservative as possible while still avoiding false positives. Since our policy is based on pointer injection, we use two tag bits per word of memory and hardware register. The *taint (T)* bit is set for untrusted data, and propagates on all arithmetic, logical, and data movement instructions. Any instruction with

| Operation | Example | Meaning | Taint Propagation | Pointer Propagation |
|---|---|---|---|---|
| Load | ld r1+imm, r2 | r2 = M[r1+imm] | T[r2] = T[M[r1+imm]] | P[r2] = P[M[r1+imm]] |
| Store | st r2, r1+imm | M[r1+imm] = r2 | T[M[r1+imm]] = T[r2] | P[M[r1+imm]] = P[r2] |
| Add/Subtract/Or | add r1, r2, r3 | r3 = r1 + r2 | T[r3] = T[r1] ∨ T[r2] | P[r3] = P[r1] ∨ P[r2] |
| And | and r1, r2, r3 | r3 = r1 ∧ r2 | T[r3] = T[r1] ∨ T[r2] | P[r3] = P[r1] ⊕ P[r2] |
| All other ALU | xor r1,r2,r3 | r3 = r1 ⊕ r2 | T[r3] = T[r2] ∨ T[r1] | P[r3] = 0 |
| Sethi | sethi imm, r1 | r1 = imm | T[r1] = 0 | P[r1] = P[insn] |
| Jump | jmpl r1+imm, r2 | r2 = pc; pc = r1 + imm | T[r2] = 0 | P[r2] = 1 |

Table 1: The DIFT propagation rules for the taint and pointer bit. T[x] and P[x] refer to the taint (T) or pointer (P) tag bits respectively for memory location, register, or instruction x.

a tainted source operand propagates taint to the destination operand (register or memory). The *pointer (P)* bit is initialized for legitimate application pointers and propagates during valid pointer operations such as pointer arithmetic. A security exception is thrown if a tainted instruction is fetched or if the address used in a load, store, or jump instruction is tainted and not a valid pointer. In other words, we allow a program to combine a valid pointer with an untrusted index, but not to use an untrusted pointer directly.

Our propagation rules for the P bit (Table 1) are derived from pointer operations used in real code. Any operation that could reasonably result in a valid pointer should propagate the P bit. For example, we propagate the P bit for data movement instructions such as `load` and `store`, since copying a pointer should copy the P bit as well. The `and` instruction is often used to align pointers. To model this behavior, the `and` propagation rule sets the P bit of the destination register if one source operand is a pointer, and the other is a non-pointer. Section 4.5 discusses a more conservative `and` propagation policy that results in runtime performance overhead.

The P bit propagation rule for addition and subtraction instructions is more permissive than the policy used in [19], due to false positives encountered in legitimate code of several applications. We propagate the P bit if either operand is a pointer as we encountered real-world situations where two pointers are added together. For example, the glibc function `_itoa_word()` is used to convert integers to strings. When given a pointer argument, it indexes bits of the pointer into an array of decimal characters on SPARC systems, effectively adding two pointers together.

Moreover, we have found that the `call` and `jmpl` instructions, which read the program counter (PC) into a register, must always set the P bit of their destination register. This is because assembly routines such as glibc `memcpy()` on SPARC contain optimized versions of Duff's device that use the PC as a pointer [15]. In `memcpy()`, a `call` instruction reads PC into a register and adds to it the (possibly tainted) copy length argument. The resulting value is used to jump into the mid-

| Operation | Example | Security Check |
|---|---|---|
| Load | ld r1+imm, r2 | T[r1] ∧ ¬ P[r1] |
| Store | st r2, r1+imm | T[r1] ∧ ¬ P[r1] |
| Jump | jmpl r1+imm, r2 | T[r1] ∧ ¬ P[r1] |
| Instruction fetch | - | T[insn] |

Table 2: The DIFT check rules for BOF detection. `rx` means register x. A security exception is raised if the condition in the rightmost column is true.

dle of a large block of copy statements. Unless the `call` and `jmpl` set the destination P bit, this behavior would cause a false positive. Similar logic can be found in the `memcmp()` function in glibc for x86 systems.

Finally, we must propagate the P bit for instructions that may initialize a pointer to a valid address in statically allocated memory. The only instruction used to initialize a pointer to statically allocated memory is `sethi`. The `sethi` instruction sets the most significant 22 bits of a register to the value of its immediate operand and clears the least significant 10 bits. If the analysis described in Section 4.2.2 determines that a `sethi` instruction is a pointer initialization statement, then the P bit for this instruction is set at process startup. We propagate the P bit of the `sethi` instruction to its destination register at runtime. A subsequent `or` instruction may be used to initialize the least significant 10 bits of a pointer, and thus must also propagate the P bit of its source operands.

The remaining ALU operations such as multiply or shift should not be performed on pointers. These operations clear the P bit of their destination operand. If a program marshals or encodes pointers in some way, such as when migrating shared state to another process [36], a more liberal pointer propagation ruleset similar to our rules for taint propagation rules may be necessary.

## 4.2 Pointer Identification

The PI-based policy depends on accurate identification of legitimate pointers in the application code in order to initialize the P bit for these memory locations. When a pointer is assigned a value derived from an existing pointer, tag propagation will ensure that the P bit is set appropriately. The P bit must only be initialized for *root pointer assignments*, where a pointer is set to a valid memory address that is not derived from another pointer. We distinguish between *static* root pointer assignments, which initialize a pointer with a valid address in statically allocated memory (such as the address of a global variable), and *dynamic* root pointer assignments, which initialize a pointer with a valid address in dynamically allocated memory.

### 4.2.1 Pointers to Dynamically Allocated Memory

To allocate memory at runtime, user code must use a system call. On a Linux SPARC system, there are five memory allocation system calls: `mmap`, `mmap2`, `brk`, `mremap`, and `shmat`. All pointers to dynamically allocated memory are derived from the return values of these system calls. We modified the Linux kernel to set the P bit of the return value for any successful memory allocation system call. This allows all dynamic root pointer assignments to be identified without false positives or negatives. Furthermore, we also set the P bit of the stack pointer register at process startup.

### 4.2.2 Pointers to Statically Allocated Memory

All static root pointer assignments are contained in the data and code sections of an object file. The data section contains pointers initialized to statically allocated memory addresses. The code section contains instructions used to initialize pointers to statically allocated memory at runtime. To initialize the P bit for static root pointer assignments, we must scan all data and code segments of the executable and any shared libraries at startup.

When the program source code is compiled to a relocatable object file, all references to statically allocated memory are placed in the relocation table. Each relocation table entry stores the location of the memory reference, the reference type, the symbol referred to, and an optional symbol offset. For example, a pointer in the data segment initialized to *&x + 4* would have a relocation entry with type data, symbol *x*, and offset 4. When the linker creates a final executable or library image from a group of object files, it traverses the relocation table in each object file and updates a reference to statically allocated memory if the symbol it refers to has been relocated to a new address.

With access to full relocation tables, static root pointer assignments can be identified without false positives or negatives. Conceptually, we set the P bit for each instruction or data word whose relocation table entry is a reference to a symbol in statically allocated memory. However, in practice full relocation tables are not available in executables or shared libraries. Hence, we must conservatively identify statically allocated memory references without access to relocation tables. Fortunately, the restrictions placed on references to statically allocated memory by the object file format allow us to detect such references by scanning the code and data segments, even without a relocation table. The only instructions or data that can refer to statically allocated memory are those that conform to an existing relocation entry format.

Like all modern Unix systems, our prototype uses the ELF object file format [41]. Statically allocated memory references in data segments are 32-bit constants that are relocated using the R_SPARC_32 relocation entry type. Statically allocated memory references in code segments are created using a pair of SPARC instructions, `sethi` and `or`. A pair of instructions is required to construct a 32-bit immediate because SPARC instructions have a fixed 32-bit width. The `sethi` instruction initializes the most significant 22 bits of a word to an immediate value, while the `or` instruction is used to initialize the least significant 10 bits (if needed). These instructions use the R_SPARC_HI22 and R_SPARC_LO10 relocation entry types, respectively.

Even without relocation tables, we know that statically allocated memory references in the code segment are specified using a `sethi` instruction containing the most significant 22 bits of the address, and any statically allocated memory references in the data segment must be valid 32-bit addresses. However, even this knowledge would not be useful if the memory address references could be encoded in an arbitrarily complex manner, such as referring to an address in statically allocated memory shifted right by four or an address that has been logically negated. Scanning code and data segments for all possible encodings would be extremely difficult and would likely lead to many false positives and negatives. Fortunately, this situation does not occur in practice, as all major object file formats (ELF [41], a.out, PE [26], and Mach-O) restrict references to statically allocated memory to a single valid symbol in the current executable or library plus a constant offset. Figure 2 presents a few C code examples demonstrating this restriction.

Algorithm 1 summarizes our scheme initializing the P bit for static root pointer assignments without relocation tables. We scan any data segments for 32-bit values that are within the virtual address range of the current executable or shared library and set the P bit for any matches. To recognize root pointer assignments in code, we scan the code segment for `sethi` instructions. If the immediate operand of the `sethi` instruction specifies a constant within the virtual address range of the current executable or shared library, we set the P bit of the instruction. Unlike the x86, the SPARC has fixed-length

```
int x,y;
int * p = &x + 0x80000000;       // symbol + any 32-bit offset is OK
int * p = &x;                    // symbol + no offset is OK
int * p = (int) &x + (int) &y;   // cannot add two symbols, will not compile
int * p = (int) &x × 4;          // cannot multiply a symbol, will not compile
int * p = (int) &x ⊕ -1;         // cannot xor a symbol, will not compile
```

Figure 2: C code showing valid and invalid references to statically allocated memory. Variables x, y, and p are global variables.

**Algorithm 1** Pseudocode for identifying static root pointer assignments in SPARC ELF binaries.

```
procedure CHECKSTATICCODE(ElfObject o, Word * w)
    if $*w$ is a sethi instruction then
        $x \leftarrow$ extract_cst22($*w$)      ▷ extract 22 bit constant from sethi, set least significant 10 bits to zero
        if $x >= o$.obj_start and $x < o$.obj_end then
            set_p_bit($w$)
        end if
    end if
end procedure

procedure CHECKSTATICDATA(ElfObject o, Word * w)
    if $*w >= o$.obj_start and $*w < o$.obj_end then
        set_p_bit($w$)
    end if
end procedure

procedure INITSTATICPOINTER(ElfObject o)
    for all segment $s$ in $o$ do
        for all word $w$ in segment $s$ do
            if $s$ is executable then
                CheckStaticCode($o$, $w$)
            end if
            CheckStaticData($o$, $w$)      ▷ Executable sections may contain read-only data
        end for
    end for
end procedure
```

instructions, allowing for easy disassembly of all code regions.

Modern object file formats do not allow executables or libraries to contain direct references to another object file's symbols, so we need to compare possible pointer values against only the current object file's start and end addresses, rather than the start and end addresses of all executable and libraries in the process address space. This algorithm is executed *once* for the executable at startup and once for each shared library when it is initialized by the dynamic linker. As shown in Section 4.5, the runtime overhead of the initialization is negligible.

In contrast with our scheme, pointer identification in the original proposal for a PI-based policy is impractical. The scheme in [19] attempts to dynamically detect pointers by checking if the operands of any instructions used for pointer arithmetic can be valid pointers to the memory regions currently used by the program. This requires scanning the page tables for every add or subtract instruction, which is prohibitively expensive.

## 4.3 Discussion

**False positives and negatives due to P bit initialization:** Without access to the relocation tables, our scheme for root pointer identification could lead to false positives or negatives in our security analysis. If an integer in the data segment has a value that happens to correspond to a valid memory address in the current executable or shared library, its P bit will be set even though it is not a pointer. This misclassification can cause a false negative in our buffer overflow detection. A false positive in the buffer overflow protection is also possible, although we have not observed one in practice thus far. All references to statically allocated memory are restricted by the object file format to a single symbol plus a constant offset. Our analysis will fail to identify a pointer only if this offset is large enough to cause the *symbol+offset* sum to refer to an address outside of the current executable object. Such a pointer would be outside the bounds of any valid memory region in the executable and would cause a segmentation fault if dereferenced.

**DIFT tags at word granularity:** Unlike prior work [19], we use per-word tags (P and T bits) rather than per-byte tags. Our policy targets pointer corruption, and modern ABIs require pointers to be naturally aligned, 32-bit values, even on the x86 [41]. Hence, we can reduce the memory overhead of DIFT from eight bits per word to two bits per word.

As explained in Section 3, we must specify how to handle partial word writes during byte or halfword stores. These writes only update part of a memory word and must combine the new tag of the value being written to memory with the old tag of the destination memory word. The combined value is then used to update the tag of the destination memory word. For taint tracking (T bit), we OR the new T bit with the old one in memory, since we want to track taint as conservatively as possible. Writing a tainted byte will taint the entire word of memory, and writing an untainted byte to a tainted word will not untaint the word. For pointer tracking (P bit), we must balance protection and false positive avoidance. We want to allow a valid pointer to be copied byte-per-byte into a word of memory that previously held an integer and still retain the P bit. However, if an attacker overwrites a single byte of a pointer [18], that pointer should lose its P bit. To satisfy these requirements, byte and halfword store instructions always set the destination memory word's P bit to that of the new value being written, ignoring the old P bit of the destination word.

**Caching P Bit initialization:** For performance reasons, it is unwise to always scan all memory regions of the executable and any shared libraries at startup to initialize the P bit. P bit initialization results can be cached, as the pointer status of an instruction or word of data at startup is always the same. The executable or library can be scanned once, and a special ELF section containing a list of root pointer assignments can be appended to the executable or library file. At startup, the security monitor could read this ELF section, initializing the P bit for all specified addresses without further scanning.

## 4.4 Portability to Other Systems

We believe that our approach is portable to other architectures and operating systems. The propagation and check rules reflect how pointers are used in practice and for the most part are architecture neutral. However, our pointer initialization rules must be ported when moving to a new platform. Identifying dynamic root pointer assignments is OS-dependent, but requires only modest effort. All we require is a list of system calls that dynamically allocate memory.

Identifying static root pointer assignments depends on both the architecture and the object file format. We expect our analysis for static pointer initializations within data segments to work on all modern platforms. This analysis assumes that initialized pointers within the data segment are word-sized, naturally aligned variables whose value corresponds to a valid memory address within the executable. To the best of our knowledge, this assumption holds for all modern object file formats, including the dominant formats for x86 systems [26, 41].

Static root pointer assignments in code segments can be complex to identify for certain architectures. Porting to other RISC systems should not be difficult, as all RISC architectures use fixed-length instructions and provide an equivalent to `sethi`. For instance, MIPS uses the load-upper-immediate instruction to set the high 16 bits of a register to a constant. Hence, we just need to adjust Algorithm 1 to target these instructions.

However, CISC architectures such as the x86 require a different approach because they support variable-length instructions. Static root pointer assignments are performed using an instruction such as `movl` that initializes a register to a full 32-bit constant. However, CISC object files are more difficult to analyze, as precisely disassembling a code segment with variable-length instructions is undecidable. To avoid the need for precise disassembly, we can conservatively identify potential instructions that contain a reference to statically allocated memory.

A conservative analysis to perform P bit initialization on CISC architectures would first scan the entire code segment for valid references to statically allocated memory. A valid 32-bit memory reference may begin at any byte in the code segment, as a variable-length ISA places no alignment restrictions on instructions. For each valid memory reference, we scan backwards to determine if any of the bytes preceding the address can form a valid instruction. This may require scanning a small number of bytes up to the maximum length of an ISA instruction. Disassembly may also reveal multiple candidate instructions for a single valid address. We examine each candidate instruction and conservatively set the P bit if the instruction may initialize a register to the valid address. This allows us to conservatively identify all static root pointer assignments, even without precise disassembly.

## 4.5 Evaluation

To evaluate our security scheme, we implemented our DIFT policy for buffer overflow prevention on the Raksha system. We extended a Linux 2.6.21.1 kernel to set

| Program | Vulnerability | Attack Detected |
|---|---|---|
| polymorph [35] | Stack overflow | Overwrite frame pointer, return address |
| atphttpd [1] | Stack overflow | Overwrite frame pointer, return address |
| sendmail [24] | BSS overflow | Overwrite application data pointer |
| traceroute [42] | Double free | Overwrite heap metadata pointer |
| nullhttpd [30] | Heap overflow | Overwrite heap metadata pointer |

Table 3: The security experiments for BOF detection in userpace.

| Program | PI (normal) | PI (`and` emulation) |
|---|---|---|
| 164.gzip | 1.002x | 1.320x |
| 175.vpr | 1.001x | 1.000x |
| 176.gcc | 1.000x | 1.065x |
| 181.mcf | 1.000x | 1.010x |
| 186.crafty | 1.000x | 1.000x |
| 197.parser | 1.000x | 2.230x |
| 254.gap | 1.000x | 2.590x |
| 255.vortex | 1.000x | 1.130x |
| 256.bzip2 | 1.000x | 1.050x |
| 300.twolf | 1.000x | 1.010x |

Table 4: Normalized execution time after the introduction of the PI-based buffer overflow protection policy. The execution time without the security policy is 1.0. Execution time higher than 1.0 represents performance degradation.

the P bit for pointers returned by memory allocation system calls and to initialize taint bits. Policy configuration registers and register tags are saved and restored during traps and interrupts. We taint the environment variables and program arguments when a process is created, and also taint any data read from the filesystem or network. The only exception is reading executable files owned by root or a trusted user. The dynamic linker requires root-owned libraries and executables to be untainted, as it loads pointers and executes code from these files.

Our security monitor initializes the P bit of each library or executable in the user's address space and handles security exceptions. The monitor was compiled as a statically linked executable. The kernel loads the monitor into the address space of every application, including `init`. When a process begins execution, control is first transferred to the monitor, which performs P bit initialization on the application binary. The monitor then sets up the policy configuration registers with the buffer overflow prevention policy, disables trusted mode, and transfers control to the real application entry point. The dynamic linker was slightly modified to call back to the security monitor each time a new library is loaded, so that P bit initialization can be performed. All application and library instructions in all userspace programs run with buffer overflow protection.

No userspace applications or libraries, excluding the dynamic linker, were modified to support DIFT analysis. All binaries in our experiments are stripped, and contain no debugging information or relocation tables. The security of our system was evaluated by attempting to exploit a wide range of buffer overflows on vulnerable, unmodified applications. The results are presented in Table 3. We successfully prevented both control and data pointer overwrites on the stack, heap, and BSS. In the case of polymorph, we also tried to corrupt a single byte or a halfword of the frame pointer instead of the whole word. Our policy detected the attack correctly as we do track partial pointer overwrites (see Section 4.3).

To test for false positives, we ran a large number of real-world workloads such as compiling applications like Apache, booting the Gentoo Linux distribution, and running Unix binaries such as perl, GCC, make, sed, awk, and ntp. No false positives were encountered, despite our conservative tainting policy.

To evaluate the performance overhead of our policy, we ran 10 integer benchmarks from the SPECcpu2000 suite. Table 4 (column titled "PI (normal)") shows the overall runtime overhead introduced by our security scheme, assuming no caching of the P bit initialization. The runtime overhead is negligible ($<$0.1%) and solely due to the initialization of the P bit. The propagation and check of tag bits is performed in hardware at runtime and has no performance overhead [13].

We also evaluated the more restrictive P bit propagation rule for `and` instructions from [19]. The P bit of the destination operand is set only if the P bit of the source operands differ, and the non-pointer operand has its sign bit set. The rationale for this is that a pointer will be aligned by masking it with a negative value, such as masking against $-4$ to force word alignment. If the user is attempting to extract a byte from the pointer – an operation which does not create a valid pointer, the sign bit of the mask will be cleared.

This more conservative rule requires any `and` instruction with a pointer argument to raise a security exception, as the data-dependent tag propagation rule is too expensive to support in hardware. The security exception handler performs this propagation in software for `and` instructions with valid pointer operands. While we encountered no false positives with this rule, performance overheads of up to 160% were observed for some SPECcpu2000 benchmarks (see rightmost column in Table 4).

We believe this stricter `and` propagation policy provides a minor improvement in security and does not justify the increase in runtime overhead.

## 5 Extending BOF Protection to Kernelspace

The OS kernel presents unique challenges for buffer overflow prevention. Unlike userspace, the kernel shares its address space with many untrusted processes, and may be entered and exited via traps. Hardcoded constant addresses are used to specify the beginning and end of kernel memory maps and heaps. The kernel may also legitimately dereference untrusted pointers in certain cases. Moreover, the security requirements for the kernel are higher as compromising the kernel is equivalent to compromising all applications and user accounts.

In this section, we extend our userspace buffer overflow protection to the OS kernel. We demonstrate our approach by using the PI-based policy to prevent buffer overflows in the Linux kernel. In comparison to prior work [11], we do not require the operating system to be ported to a new architecture, protect the entire OS codebase with no real-world false positives or errors, support self-modifying code, and have low runtime overhead. We also provide the first comprehensive runtime detection of user-kernel pointer dereference attacks.

### 5.1 Entering and Exiting Kernelspace

The tag propagation and check rules described in Tables 1 and 2 for userspace protection are also used with the kernel. The kernelspace policy differs only in the P and T bit initialization and the rules used for handling security exceptions due to tainted pointer dereferences.

Nevertheless, the system may at some point use different security policies for user and kernel code. To ensure that the proper policy is applied to all code executing within the operating system, we take advantage of the fact that the only way to enter the kernel is via a trap, and the only way to exit is by executing a return from trap instruction. When a trap is received, trusted mode is enabled by hardware and the current policy configuration registers are saved to the kernel stack by the trap handler. The policy configuration registers are then reinitialized to the kernelspace buffer overflow policy and trusted mode is disabled. Any subsequent code, such as the actual trap handling code, will now execute with kernel BOF protection enabled. When returning from the trap, the configuration registers for the interrupted user process must be restored.

The only kernel instructions that do not execute with buffer overflow protection enabled are the instructions that save and restore configuration registers during trap entry and exit, a few trivial trap handlers written in assembly which do not access memory at all, and the fast path of the SPARC register window overflow/underflow handler. We do not protect these handlers because they do not use a runtime stack and do not access kernel memory unsafely. Enabling and disabling protection when entering and exiting such handlers could adversely affect system performance without improving security.

### 5.2 Pointer Identification in the Presence of Hardcoded Addresses

The OS kernel uses the same static root pointer assignment algorithm as userspace. At boot time, the kernel image is scanned for static root pointer assignments by scanning its code and data segments, as described in Section 4. However, dynamic root pointer assignments must be handled differently. In userspace applications, dynamically allocated memory is obtained via OS system calls such as `mmap` or `brk`. In the operating system, a variety of memory map regions and heaps are used to dynamically allocate memory. The start and end virtual addresses for these memory regions are specified by hardcoded constants in kernel header files. All dynamically allocated objects are derived from the hardcoded start and end addresses of these dynamic memory regions.

In kernelspace, all dynamic root pointer assignments are contained in the kernel code and data at startup. When loading the kernel at system boot time, we scan the kernel image for references to dynamically allocated memory maps and heaps. All references to dynamically allocated memory must be to addresses within the kernel heap or memory map regions identified by the hardcoded constants. To initialize the P bit for dynamic root pointer assignments, any `sethi` instruction in the code segment or word of data in the data segment that specifies an address within one of the kernel heap or memory map regions will have its P bit set. Propagation will then ensure that any values derived from these pointers at runtime will also be considered valid pointers. The P bit initialization for dynamic root pointer assignments and the initialization for static root pointer assignments can be combined into a single pass over the code and data segments of the OS kernel image at bootup.

On our Linux SPARC prototype, the only heap or memory map ranges that should be indexed by untrusted information are the $vmalloc$ heap and the fixed address, $pkmap$, and $srmmu - nocache$ memory map regions. The start and end values for these memory regions can be easily determined by reading the header files of the operating system, such as the $vaddrs$ SPARC-dependent header file in Linux. All other memory map and and heap regions in the kernel are small private I/O memory map regions whose pointers should never be indexed by untrusted information and thus do not need to be identified during P bit initialization to prevent false positives.

Kernel heaps and memory map regions have an inclusive lower bound, but exclusive upper bound. However, we encountered situations where the kernel would compute valid addresses relative to the upper bound. In this situation, a register is initialized to the upper bound of a memory region. A subsequent instruction subtracts a non-zero value from the register, forming a valid address within the region. To allow for this behavior, we treat a `sethi` constant as a valid pointer if its value is greater than or equal to the lower bound of a memory region and less than or equal to the upper bound of a memory region, rather than strictly less than the upper bound. This issue was never encountered in userspace.

## 5.3 Untrusted Pointer Dereferences

Unlike userspace code, there are situations where the kernel may legitimately dereference an untrusted pointer. Many OS system calls take untrusted pointers from userspace as an argument. For example, the second argument to the `write` system call is a pointer to a user buffer.

Only special routines such as `copy_to_user()` in Linux or `copyin()` in BSD may safely dereference a userspace pointer. These routines typically perform a simple bounds check to ensure that the user pointer does not point into the kernel's virtual address range. The untrusted pointer can then safely be dereferenced without compromising the integrity of the OS kernel. If the kernel does not perform this access check before dereferencing a user pointer, the resulting security vulnerability allows an attacker to read or write arbitrary kernel addresses, resulting in a full system compromise.

We must allow legitimate dereferences of tainted pointers in the kernel, while still preventing pointer corruption from buffer overflows and detecting unsafe user pointer dereferences. Fortunately, the design of modern operating systems allows us to distinguish between legitimate and illegitimate tainted pointer dereferences. In the Linux kernel and other modern UNIX systems, the only memory accesses that should cause an MMU fault are accesses to user memory. For example, an MMU fault can occur if the user passed an invalid memory address to the kernel or specified an address whose contents had been paged to disk. The kernel must distinguish between MMU faults due to load/stores to user memory and MMU faults due to bugs in the OS kernel. For this purpose, Linux maintains a list of all kernel instructions that can access user memory and recovery routines that handle faults for these instructions. This list is kept in the special ELF section $\_\_ex\_table$ in the Linux kernel image. When an MMU fault occurs, the kernel searches $\_\_ex\_table$ for the faulting instruction's address. If a match is found, the appropriate recovery routine is called. Otherwise, an operating system bug has occurred and the kernel panics.

We modified our security handler so that on a security exception due to a load or store to an untrusted pointer, the memory access is allowed if the program counter (PC) of the faulting instruction is found in the $\_\_ex\_table$ section and the load/store address does not point into kernelspace. Requiring tainted pointers to specify userspace addresses prevents user/kernel pointer dereference attacks. Additionally, any attempt to overwrite a kernel pointer using a buffer overflow attack will be detected because instructions that access the corrupted pointer will not be found in the $\_\_ex\_table$ section.

## 5.4 Portability to Other Systems

We believe this approach is portable to other architectures and operating systems. To perform P bit initialization for a new operating system, we would need to know the start and end addresses of any memory regions or heaps that would be indexed by untrusted information. Alternatively, if such information was unavailable, we could consider any value within the kernel's virtual address space to be a possible heap or memory map pointer when identifying dynamic root pointer assignments at system bootup.

Our assumption that MMU faults within the kernel occur only when accessing user addresses also holds for FreeBSD, NetBSD, OpenBSD, and OpenSolaris. Rather than maintaining a list of instructions that access user memory, these operating systems keep a special MMU fault recovery function pointer in the Process Control Block (PCB) of the current task. This pointer is only non-NULL when executing routines that may access user memory, such as `copyin()`. If we implemented our buffer overflow protection for these operating systems, a tainted load or store would be allowed only if the MMU fault pointer in the PCB of the current process was non-NULL and the load or store address did not point into kernelspace.

## 5.5 Evaluation

To evaluate our buffer overflow protection scheme with OS code, we enabled our PI policy for the Linux kernel. The SPARC BIOS was extended to initialize the P bit for the OS kernel at startup. After P bit initialization, the BIOS initializes the policy configuration registers, disables trusted mode, and transfers control to the entry point of the OS kernel with buffer overflow protection enabled.

| Module Targeted | Vulnerability | Attack Detected |
|---|---|---|
| quotactl system call [52] | User/kernel pointer | Tainted pointer to kernelspace |
| i2o driver [52] | User/kernel pointer | Tainted pointer to kernelspace |
| sendmsg system call [2, 47] | Heap overflow<br>Stack Overflow | Overwrite heap metadata pointer<br>Overwrite local data pointer |
| moxa driver [45] | BSS Overflow | Overwrite BSS data pointer |
| cm4040 driver [40] | Heap Overflow | Overwrite heap metadata pointer |

Table 5: The security experiments for BOF detection in kernelspace.

When running the kernel, we considered any data received from the network or disk to be tainted. Any data copied from userspace was also considered tainted, as were any system call arguments from a userspace system call trap. As specified in Section 4.5, we also save/restore policy registers and register tags during traps. The above modifications were the only changes made to the kernel. All other code, even optimized assembly copy routines, context switching code, and bootstrapping code at startup, were left unchanged and ran with buffer overflow protection enabled. Overall, our extensions added 1774 lines to the kernel and deleted 94 lines, mostly in architecture-dependent assembly files. Our extensions include 732 lines of code for the security monitor, written in assembly.

To evaluate the security of our approach, we exploited real-world user/kernel pointer dereference and buffer overflow vulnerabilities in the Linux kernel. Our results are summarized in Table 5. The sendmsg vulnerability allows an attacker to choose between overwriting a heap buffer or stack buffer. Our kernel security policy was able to prevent all exploit attempts. For device driver vulnerabilities, if a device was not present on the FPGA-based prototype, we simulated sufficient device responses to reach the vulnerable section of code and perform our exploit.

We evaluated the issue of false positives by running the kernel with our security policy enabled under a number of system call-intensive workloads. We compiled large applications from source, booted Gentoo Linux, performed logins via OpenSSH, and served web pages with Apache. Despite our conservative tainting policy, we encountered only one issue, which initially seemed to be a false positive. However, we have established it to be a bug and potential security vulnerability in the current Linux kernel on SPARC32 and have notified the Linux kernel developers. This issue occurred during the `__bzero()` routine, which dereferenced a tainted pointer whose address was not found in the $\_\_ex\_table$ section. As user pointers may be passed to `__bzero()`, all memory operations in `__bzero()` should be in $\_\_ex\_table$. Nevertheless, a solitary block of store instructions did not have an entry. A malicious user could potentially exploit this bug to cause a local denial-of-service attack, as any MMU faults caused by these stores would cause a kernel panic. After fixing this bug by adding the appropriate entry to $\_\_ex\_table$, no further false positives were encountered in our system.

Performance overhead is negligible for most workloads. However, applications that are dominated by copy operations between userspace and kernelspace may suffer noticeable slowdown, up to 100% in the worst case scenario of a file copy program. This is due to runtime processing of tainted user pointer dereferences, which require the security exception handler to verify the tainted pointer address and find the faulting instruction in the $\_\_ex\_table$ section.

We profiled our system and determined that almost all of our security exceptions came from a single kernel function, $copy\_user()$. To eliminate this overhead, we manually inserted security checks at the beginning of $copy\_user()$ to validate any tainted pointers. After the input is validated by our checks, we disable data pointer checks until the function returns. This change reduced our performance overhead to a negligible amount (<0.1%), even for degenerate cases such as copying files. Safety is preserved, as the initial checks verify that the arguments to this function are safe, and manual inspection of the code confirmed that $copy\_user()$ would never behave unsafely, so long as its arguments were validated. Our control pointer protection prevents attackers from jumping into the middle of this function. Moreover, while checks are disabled while $copy\_user()$ is executing, taint propagation is still on. Hence, $copy\_user()$ cannot be used to sanitize untrusted data.

## 6 Comprehensive Protection with Hybrid DIFT Policies

The PI-based policy presented in this paper prevents attackers from corrupting any code or data pointers. However, false negatives do exist, and limited forms of memory corruption attacks may bypass our protection. This should not be surprising, as our policy focuses on a specific class of attacks (pointer overwrites) and operates on unmodified binaries without source code access. In this section, we discuss security policies that can be used to mitigate these weaknesses.

False negatives can occur if the attacker overwrites non-pointer data without overwriting a pointer [6]. This is a limited form of attack, as the attacker must use a buffer overflow to corrupt non-pointer data without corrupting any pointers. The application must then use the corrupt data in a security-sensitive manner, such as an array index or a flag determining if a user is authenticated. The only form of non-pointer overwrite our PI policy detects is code overwrites, as tainted instruction execution is forbidden. Non-pointer data overwrites are not detected by our PI policy and must be detected by a separate, complementary buffer overflow protection policy.

## 6.1 Preventing Pointer Offset Overwrites

The most frequent way that non-pointers are used in a security-sensitive manner is when an integer is used as an array index. If an attacker can corrupt an array index, the next access to the array using the corrupt offset will be attacker-controlled. This indirectly allows the attacker to control a pointer value. For example, if the attacker wants to access a memory address `y` and can overwrite an index into array `x`, then the attacker should overwrite the index with the value `y−x`. The next access to `x` using the corrupt index will then access `y` instead.

Our PI policy does not prevent this attack because no pointer was overwritten. We cannot place restrictions on array indices or other type of offsets without bounds information or bounds check recognition. Without source code access or application-specific knowledge, it is difficult to formulate general rules to protect non-pointers without false positives. If source code is available, the compiler may be able to automatically identify security-critical data, such as array offsets and authentication flags, that should never be tainted [4].

A recently proposed form of ASLR [20] can be used to protect against pointer offset overwrites. This novel ASLR technique randomizes the relative offsets between variables by permuting the order of variables and functions *within* a memory region. This approach would probabilistically prevent all data and code pointer offset overwrites, as the attacker would be unable to reliably determine the offset between any two variables or functions. However, randomizing relative offsets requires access to full relocation tables and may not be backwards compatible with programs that use hardcoded addresses or make assumptions about the memory layout. The remainder of this section discusses additional DIFT policies to prevent non-pointer data overwrites without the disadvantages of ASLR.

## 6.2 Protecting Offsets for Control Pointers

To the best of our knowledge, only a handful of reported vulnerabilities allow control pointer offsets to be overwritten [25, 50]. This is most likely due to the relative infrequency of large arrays of function pointers in real-world code. A buffer overflow is far more likely to directly corrupt a pointer before overwriting an index into an array of function pointers.

Nevertheless, DIFT platforms can provide control pointer offset protection by combining our PI-based policy with a restricted form of BR-based protection. If BR-based protection is only used to protect control pointers, then the false positive issues described in Section 2.2 do not occur in practice [10]. To verify this, we implemented a control pointer-only BR policy and applied the policy to userspace and kernelspace. This policy did not result in any false positives, and prevented buffer overflow attacks on control pointers. Our policy classified `and` instructions and all comparisons as bounds checks.

The BR policy has false negatives in different situations than the PI policy. Hence the two policies are complementary. If control-pointer-only BR protection and PI protection are used concurrently, then a false negative would have to occur in both policies for a control pointer offset attack to succeed. The attacker would have to find a vulnerability that allowed a control pointer offset to be corrupted without corrupting a pointer. The application would then have to perform a comparison instruction or an `and` instruction that was not a real bounds check on the corrupt offset before using it. We believe this is very unlikely to occur in practice. As we have observed no false positives in either of these policies, even in kernelspace, we believe these policies should be run concurrently for additional protection.

## 6.3 Protecting Offsets for Data Pointers

Unfortunately, the BR policy cannot be applied to data pointer offsets due to the severe false positive issues discussed in Section 1. However, specific situations may allow for DIFT-based protection of non-pointer data. For example, Red Zone heap protection prevents heap buffer overflows by placing a canary or special DIFT tag at the beginning of each heap chunk [37, 39]. This prevents heap buffer overflows from overwriting the next chunk on the heap and also protects critical heap metadata such as heap object sizes.

Red Zone protection can be implemented by using DIFT to tag heap metadata with a sandboxing bit. Access to memory with the sandboxing bit set is forbidden, but sandboxing checks are temporarily disabled when `malloc()` is invoked. A modified `malloc()` is necessary to maintain the sandboxing bit, setting it for newly created heap metadata and clearing it when a heap metadata block is freed. The DIFT Red Zone heap protection could be run concurrently with PI protection, providing enhanced protection for non-pointer data on the heap.

We implemented a version of Red Zone protection that forbids heap metadata from being overwritten, but allows out-of-bounds reads. We then applied this policy to both glibc $malloc()$ in userspace and the Linux slab allocator in kernelspace. No false positives were encountered during any of our stress tests, and we verified that all of our heap exploits from our userspace and kernelspace security experiments were detected by the Red Zone policy.

## 6.4 Beyond Pointer Corruption

Not all memory corruption attacks rely on pointer or pointer offset corruption. For example, some classes of format string attacks use only untainted pointers and integers [12]. While these attacks are rare, we should still strive to prevent them. Previous work on the Raksha system provides comprehensive protection against format string attacks using a DIFT policy [13]. The policy uses the same taint information as our PI buffer overflow protection. All calls to the `printf()` family of functions are interposed on by the security monitor, which verifies that the format string does not contain tainted format string specifiers such as %n.

For the most effective memory corruption protection for unmodified binaries, DIFT platforms such as Raksha should concurrently enable PI protection, control-pointer-only BR protection, format string protection, and Red Zone heap protection. This would prevent pointer and control pointer offset corruption and provide complete protection against format string and heap buffer overflow attacks. We can support all these policies concurrently using the four tag bits provided by the Raksha hardware. The P bit and T bit are used for buffer overflow protection and the T bit is also used to track tainted data for format string protection. The sandboxing bit, which prevents stores or code execution from tagged memory locations, is used to protect heap metadata for Red Zone bounds checking, to interpose on calls to the `printf()` functions, and to protect the security monitor (see Section 3.1). Finally, the fourth tag bit is used for control-pointer-only BR protection.

## 7 Conclusions

We presented a robust technique for buffer overflow protection using DIFT to prevent pointer overwrites. In contrast to previous work, our security policy works with unmodified binaries without false positives, prevents both data and code pointer corruption, and allows for practical hardware support. Moreover, this is the first security policy that provides robust buffer overflow prevention for the kernel and dynamically detects user/kernel pointer dereferences.

To demonstrate our proposed technique, we implemented a full-system prototype that includes hardware support for DIFT and a software monitor that manages the security policy. The resulting system is a full Gentoo Linux workstation. We show that our prototype prevents buffer overflow attacks on applications and the operating system kernel without false positives and has an insignificant effect on performance. The full-system prototyping approach was critical in identifying and addressing a number of practical issues that arise in large user programs and in the kernel code.

There are several opportunities for future research. We plan to experiment further with the concurrent use of complementary policies that prevent overwrites of non-pointer data (see Section 6). Another promising direction is applying taint rules to system call arguments. Per-application system call rules could be learned automatically, restricting tainted arguments to security-sensitive system calls such as opening files and executing programs.

## Acknowledgments

We would like to thank Jiri Gaisler, Richard Pender, and Gaisler Research for their invaluable assistance with our prototype development. We would also like to thank Anil Somayaji and the anonymous reviewers for their feedback. This work was supported NSF Awards number 0546060 and 0701607 and Stanford Graduate Fellowships by Sequoia Capital and Cisco Systems.

## References

[1] ATPHTTPD Buffer Overflow Exploit Code. `http://www.securiteam.com/exploits/6B00K003GY.html`, 2001.

[2] Attacking the Core: Kernel Exploiting Notes. `http://phrack.org/issues.html?issue=64\&id=6`, 2007.

[3] Bypassing PaX ASLR protection. `http://www.phrack.org/issues.html?issue=59\&id=9`, 2002.

[4] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th conference on Operating Systems Design and Implementation*, 2006.

[5] S. Chen, J. Xu, et al. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the Intl. Conference on Dependable Systems and Networks*, 2005.

[6] S. Chen, J. Xu, et al. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[7] J. Chung, M. Dalton, et al. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *Proceedings of the 14th Intl. Symposium on High-Performance Computer Architecture*, 2008.

[8] Computer Emergency Response Team. "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. `http:`

//www.cert.org/advisories/CA-2001-19.html, 2002.

[9] C. Cowan, C. Pu, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[10] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Intl. Symposium on Microarchitecture*, 2004.

[11] J. Criswell, A. Lenharth, et al. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st Symposium on Operating System Principles*, Oct. 2007.

[12] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[13] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Intl. Symposium on Computer Architecture*, 2007.

[14] M. Dowd. Application-Specific Attacks: Leveraging the ActionScript Virtual Machine. http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf, 2008.

[15] Duff's Device. http://www.lysator.liu.se/c/duffs-device.html, 1983.

[16] H. Etoh. GCC Extension for Protecting Applications from Stack-smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/.

[17] P. Ferrie. ANI-hilate This Week. In *Virus Bulletin*, Mar. 2007.

[18] The frame pointer overwrite. http://www.phrack.org/issues.html?issue=55\&id=8, 1999.

[19] S. Katsunuma, H. Kuriyta, et al. Base Address Recognition with Data Flow Tracking for Injection Attack Detection. In *Proceedings of the 12th Pacific Rim Intl. Symposium on Dependable Computing*, 2006.

[20] C. Kil, J. Jun, et al. Address Space Layout Permutation: Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of 22nd Applied Computer Security Applications Conference*, 2006.

[21] R. Lee, D. Karig, et al. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the Intl. Conference on Security in Pervasive Computing*, 2003.

[22] LEON3 SPARC Processor. http://www.gaisler.com.

[23] Linux Kernel Remote Buffer Overflow Vulnerabilities. http://secwatch.org/advisories/1013445/, 2006.

[24] M. Dowd. Sendmail Header Processing Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/6991.

[25] Microsoft Excel Array Index Error Remote Code Execution. http://lists.virus.org/bugtraq-0607/msg00145.html.

[26] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*, 2006.

[27] D. Moore, V. Paxson, et al. Inside the Slammer Worm. *IEEE Security & Privacy*, 23(4), July 2003.

[28] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[29] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[30] Netric Security Team. Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774, 2002.

[31] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[32] A. Nguyen-Tuong, S. Guarnieri, et al. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.

[33] OpenBSD IPv6 mbuf Remote Kernel Buffer Overflow. http://www.securityfocus.com/archive/1/462728/30/0/threaded, 2007.

[34] The PaX project. http://pax.grsecurity.net.

[35] Polymorph Filename Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/7663, 2003.

[36] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.

[37] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Intl. Symposium on High-Performance Computer Architecture*, 2005.

[38] F. Qin, C. Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th the Intl. Symposium on Microarchitecture*, 2006.

[39] W. Robertson, C. Kruegel, et al. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th Large Installation System Administration Conference*, 2003.

[40] D. Roethlisberger. Omnikey Cardman 4040 Linux Drivers Buffer Overflow. http://www.securiteam.com/unixfocus/5CP0D0AKUA.html, 2007.

[41] Santa Cruz Operation. *System V Application Binary Interface, 4th ed.*, 1997.

[42] P. Savola. LBNL Traceroute Heap Corruption Vulnerability. http://www.securityfocus.com/bid/1739, 2000.

[43] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls. In *Proceedings of the 14th ACM Conference on Computer Security*, 2007.

[44] H. Shacham, M. Page, et al. On the Effectiveness of Address Space Randomization. In *Proceedings of the 11th ACM Conference on Computer Security*, 2004.

[45] B. Spengler. Linux Kernel Advisories. http://lwn.net/Articles/118251/, 2005.

[46] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow, 2004.

[47] A. Viro. Linux Kernel Sendmsg() Local Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/14785, 2005.

[48] Microsoft Windows TCP/IP IGMP MLD Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/27100, 2008.

[49] O. Whitehouse. GS and ASLR in Windows Vista, 2007.

[50] F. Xing and B. Mueller. Macromedia Flash Player Array Index Overflow. http://securityvulns.com/Fnews426.html.

[51] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[52] J. Yang. Potential Dereference of User Pointer Errors. http://lwn.net/Articles/26819/, 2003.

## THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see http://www.usenix.org.

## SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

## Thanks to USENIX & SAGE Corporate Supporters

### USENIX Patrons

Google Microsoft Research NetApp

### USENIX Benefactors

hp invent IBM LINUX PRO MAGAZINE vmware

### USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
FOTO SEARCH Stock Footage and Stock Photography
Raytheon
Splunk
Zenoss

### USENIX Partners

Cambridge Computer Services, Inc.
GroundWork Open Source Solutions
Hyperic
Infosys
Intel
Oracle
Ripe NCC
Sendmail, Inc.
Sun Microsystems, Inc.

### SAGE Partner

MSB Associates

ISBN-13: 978-1-931971-60-7
90000
9 781931 971607